UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Implementación de redes neuronales en sistemas empotrados de altas prestaciones

Autor: Francisco Blanco Esquivel
Tutor: Gustavo Sutter

octubre 2020

Francisco Blanco Esquivel
Implementación de redes neuronales en sistemas empotrados de altas prestaciones

Francisco Blanco Esquivel
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mi familia

Jugamos como siempre
y perdimos como nunca.

Froilán Giganto

Agradecimientos

Quiero comenzar agradeciendo a mi tutor, Gustavo Sutter, por cederme los medios para poder llevar a cabo este trabajo, por guiarme en estos temas en un principio desconocidos para mi y escuchar siempre con entusiasmo mis propuestas. Agradecer también a mis padres por priorizar siempre la educación y el saber estar. Por ultimo agradecer a todo aquel que se ha interesado por la fecha de entrega de este trabajo, amigos y todo el que haya expresado sorpresa por la absurda cantidad de tiempo invertido.

Resumen

Las redes neuronales han sido uno de los grandes protagonistas de la informática esta década. Cada año el número de artículos científicos sobre aprendizaje automático aumenta exponencialmente superando en crecimiento incluso a la propia ley de Moore. Irónicamente es la mejora en la capacidad de cómputo lo que ha permitido todos estos avances en primer lugar. Pero si algo aumenta exponencialmente cada año es el número de gente que clama que la ley de Moore ha muerto y esperar a que un algoritmo sea mágicamente viable porque se ha triplicado la cantidad de transistores por mm^2 no es una opción. Esto ha impulsado a numerosas empresas, entre ellas Xilinx, a apostar por el diseño de hardware especifico en conjunto con el diseño de algoritmos. Aplicaciones en tiempo real como la conducción autónoma o robótica asistencial se benefician especialmente de estos avances al aumentar la eficiencia energética y reducir la latencia de sus implementaciones.

En este trabajo se exponen diferentes técnicas tanto del lado del diseño de hardware como del algorítmico. Con un enfoque en las redes neuronales convolucionales, las cuales han revolucionado la visión por computador. No solo en el ámbito de la computación en la nube, sino también en la computación en el Edge, ya que ofrecen muchas posibilidades de optimización y paralelización. Sin el uso de sistemas empotrados muchas aplicaciones con requisitos muy estrictos en cuanto a rendimiento, resistencia a fallos y eficiencia energética no serian posibles ya que los procesadores de propósito general solo suelen cumplir uno de estos requisitos simultáneamente. El desafió es entonces encajar estas redes neuronales en un sistema tan limitado como puede ser uno empotrado.

Palabras clave

Redes convolucionales, visión por computador, edge computing, FPGA

Abstract

Neural networks are one of the big protagonist in computer science this decade. Each year the number of papers about machine learning grows exponentially, even surpassing Moore's Law. Ironically is Moore's Law what pushed computing power to the point where these algorithms became viable. But if something has grown exponentially, it must be the number of people declaring Moore's Law dead, which means we can no longer wait for an algorithm to magically became viable because the amount of transistors by mm^2 tripled. This has pushed some companies, such as Xilinx, to invest on specific hardware design alongside algorithm development. Real time applications like autonomous driving or assistant robots benefit from this, more so given the efficiency and low latency of its implementations.

In this work different techniques from the hardware design and algorithm development are shown. With an emphasis on convolutional neural networks, which have revolutionized computer vision. Not only when it comes to cloud computing but also edge computing, given the amount of parallelism and optimization potential. Embedded systems make possible some applications which are very strict about failure resistance, power efficiency and low latency. The challenge is to fit this large convolutional networks into our limited embedded systems.

Keywords

Convolutional neural networks, computer vision, edge computing, FPGA

Índice

Lista de figuras

Lista de tablas

1

Introducción

El aprendizaje automático ha resultado ser una de las áreas que mas interés ha despertado en la informática esta última década, y con el aumento exponencial de la capacidad de computo, el aprendizaje profundo, o deep learning (DL), ha probado ser una técnica viable a la hora de atacar problemas muy complejos como el procesamiento del lenguaje natural o la clasificación de imágenes. En la actualidad son numerosas las compañías que enfocan sus esfuerzos en construir sistemas capaces de llevar a cabo estas técnicas de DL de la forma mas rápida y eficiente posible, mediante el desarrollo de hardware especifico. En este trabajo se explora el set de herramientas que proporciona Xilinx para la implementación de redes neuronales en sistemas de alto rendimiento, con un enfoque en la detección de objetos en tiempo real.

1.1. Motivación

A menudo es necesario un centro de procesamiento de datos para poder llevar a cabo tareas con redes neuronales. La motivación principal de este trabajo es explorar la implementación de sistemas mucho menos costosos los cuales están abriendo las puertas de estas tecnologías a los mundos de la automoción con el desarrollo de coches autónomos, herramientas de reconocimiento facial, etc.

Empresas como Intel, Apple y Tesla han desarrollado sus propios chips con el fin de acelerar el uso de redes neuronales profundas (Deep Neuronal Networks, DNN) para el uso en sus propios dispositivos, destinando numerosos recursos humanos y económicos para esto. Con el creciente interés en la materia Xilinx ha creado un flujo de trabajo que permite a programadores sin experiencia en desarrollo de hardware la implementación eficiente de algoritmos complejos en FPGAs. Las aplicaciones que pueden tener las FPGAs son infinitas y reducir el margen de dificultad en su implementación supone un gran empujón en muchas mas áreas de la tecnología, como ya estamos empezando a ver hoy en día.

1.2. Objetivos

Se pueden dividir los objetivos del trabajo en tres partes, primero entender el punto en el que se encuentra la visión artificial, como funcionan en detalle las redes convolucionales y que algoritmos conforman el estado del arte.

El segundo objetivo implicaría entender como nos puede ayudar una FPGA a acelerar estos algoritmos y que ventajas nos ofrece frente a CPUs y GPUs. Además de ver que cualidades de las redes convolucionales podemos explotar gracias a las FPGAs.

Finalmente mostramos la implementación de la ultima versión de unos de los algoritmos más utilizados en el reconocimiento de objetos, YOLOv4 [1], su rendimiento en un sistema de altas prestaciones y como se compara con distintas implementaciones como Resnet-50 [2] y YOLOv3 [3].

1.3. Estructura del documento

Este documento consta de 5 capítulos, siendo el primero la introducción. El segundo capitulo sirve para ubicarse dentro del estado del arte en cuanto a los principales temas de este trabajo. En una tercera parte se expone el algoritmo a implementar y los métodos de aceleración utilizados. A continuación se muestran los resultados obtenidos en cuanto al rendimiento del algoritmo y la plataforma. Y por ultimo se exponen las conclusiones y el trabajo futuro en cuanto al desarrollo del trabajo.

2

Estado del arte

En este apartado exponemos el estado actual y los últimos avances en los campos del aprendizaje profundo y su aceleración en hardware. También es importante entender que estos avances no surgen de la nada, y muchas de las técnicas usadas hoy en día se conocen desde hace décadas, y es ahora cuando tenemos la capacidad computacional para hacerlas viables.

Primero explicamos el estado del arte dentro de las redes neuronales profundas (Deep Neural Networks, DNN), mas en concreto redes convolucionales (Convolutional Neural Networks, CNN), usadas ampliamente en problemas de visión por computador. Una vez conocemos el funcionamiento de las CNNs es fácil ver como el uso de hardware especifico nos puede ayudar a disminuir tanto los tiempos de aprendizaje como de inferencia. Y veremos que acercamientos se han usado en el pasado y como han dado pie al estado actual del arte.

2.1. Redes neuronales profundas

Los términos aprendizaje automático y redes neuronales profundas se han convertido en algo de dominio público, y despiertan un gran interés en prácticamente todos los sectores profesionales como pueden ser la automoción o la medicina. Aunque parezcan conceptos extremadamente modernos, se teoriza con redes neuronales desde que en 1943 McCulloch y Pitts [4], interesados en como funciona el cerebro humano, modelan las interacciones entre neuronas como un circuito eléctrico. En 1958 nace el Perceptron de Rosenblatt [5], pavimentando el camino para toda la investigación posterior, aunque lejos de resolver problemas como el procesamiento del lenguaje natural (PLN) o el reconocimiento de objetos. Es entonces cuando en 1998 LeCun et al. [6] exploran las capacidades de las redes convolucionales en tareas como reconocimiento de patrones y reconocimiento de caracteres escritos a mano. *LeNet-5* disponía de 7 capas, y daba los mismos resultados que las Maquinas de Soporte Vectorial [7] (Support Vector Machines, SVM) con una fracción del numero de operaciones que estas requieren.

Ya a finales del siglo XX se conocían los beneficios de aumentar el numero de capas, entre ellos, mimetizar los distintos niveles de abstracción del cerebro humano. Aumentar la complejidad trae consigo un problema, el entrenamiento de estas redes requería muchas más iteraciones al aumentar conside-

rablemente el numero de pesos, además las bases de datos más populares consistían en decenas de miles de imágenes lo cual no solo hacia inviable entrenar una DNN sino que sistemas como SVMs dominaban las competiciones como PASCAL-VOC, dejando las DNNs en un segundo plano. Es entonces en 2012 cuando se publican los resultados del concurso ImageNet [8], el ganador AlexNet [9] demostró que las redes convolucionales profundas son capaces de obtener resultados récord (recortando casi la mitad del error del segundo clasificado) en tareas de clasificación de imágenes, en parte gracias al aumento en el tamaño de las bases de datos de imágenes y mejoras en la capacidad de computo.

En la actualidad las DNNs han conseguido romper numerosas barreras dentro del mundo de la Inteligecia Artificial (IA), desafíos como derrotar al actual campeón del mundo en el juego milenario de Go por AlphaGo [10], una DNN entrenada en su completitud jugando contra ella misma. Su sucesora, AlphaZero [11], esta en busca de una IA más generalista la cual sea capaz de obtener resultados sobrehumanos en numerosos juegos, entre ellos el ajedrez, shogi y StarCraft II. Además de los juegos, un tipo concreto de DNN, las redes neuronales recursivas (Recursive Neural Net, RNN) son usadas en el campo de las finanzas y PLN, y aunque estemos lejos de resolver el lenguaje, muchas empresas como Microsoft, Google o OpenAI están destinando numerosos recursos en el estudio de PLN. En este trabajo nos enfocaremos en el mundo de la visión por computador ya que es el que más se beneficia de los aceleradores, dada la naturaleza del problema y la necesidad de reducir el consumo de energía en implementaciones en el *Edge*.

2.1.1. Redes convolucionales

Las redes neuronales profundas abarcan muchas técnicas distintas, en este apartado nos enfocaremos en las CNNs y como se han convertido en el estándar dentro de la visión artificial. Desde *LeNet* que se conoce su eficiencia, con los años han ido ganando profundidad y se han refinado pero las operaciones básicas siguen siendo las mismas. Es un algoritmo de aprendizaje supervisado, la clasificación se realiza con propagación hacia delante, también llamada inferencia, y el entrenamiento con propagación hacia atrás.

En esencia una CNN consta de dos partes, la primera de extracción de características está formada por capas convolucionales y seguida de esta, una capa de clasificación formada por capas completamente conectadas (Fully Conncected Layers, FCL). Cada una de las capas convolucionales aporta un nuevo nivel de abstracción a la representación del modelo. Los filtros de las primeras capas generan mapas de características (feature maps, fmaps) de más bajo nivel, y a medida que avanzamos en el modelo los filtros son más avanzados y resaltan cualidades de la imagen más especificas. A continuación se explican los distintos elementos que pueden formar un CNN, y aunque los pesos se usen únicamente en capas convolucionales y FCL existen otras operaciones interesantes y dependiendo de la arquitectura de la red se disponen de diferente forma.

Convolución: Es la operación básica en las CNNs y además de darles nombre supone la mayoría

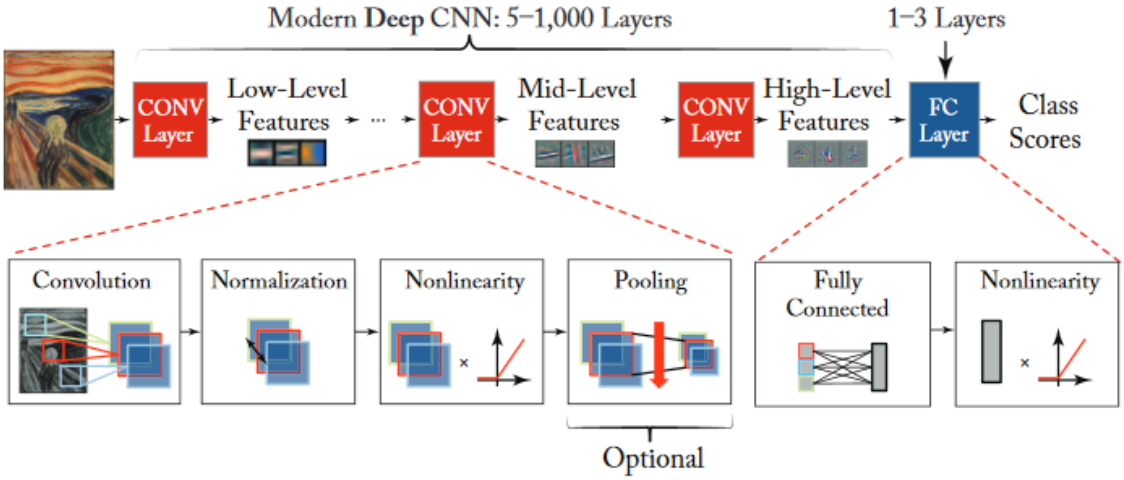

Figura 2.1: Ejemplo de arquitectura de una CNN moderna. Cada CONV layer engloba diversas operaciones, convoluciones, normalización, funciones no lineales y de forma opcional pooling. Conforme avanzamos en la red pasamos de detectar características a más alto nivel hasta que llegamos a una FCL que realiza la clasificación en un sentido más clásico.. Figura extraída de [12]

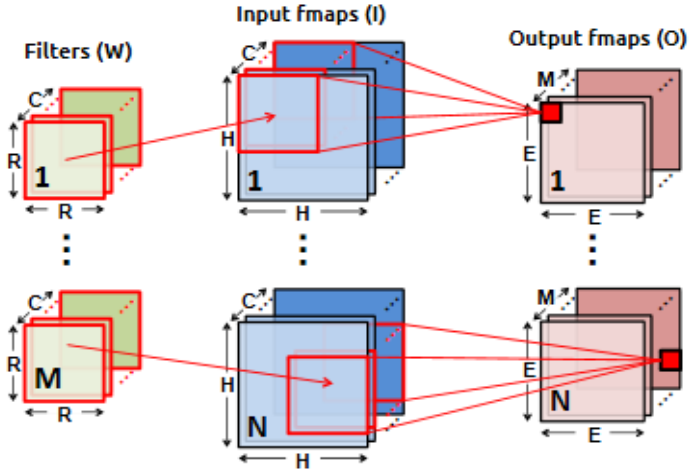

Figura 2.2: Computación llevada a cabo en la capa convolucional. Desde el primer filtro hasta el *M* para completar cada uno de los pixeles del Output feature map. *E* en la notación seleccionada corresponde con *P y Q* al ser cuadrado el fmap. Figura extraída de [13]

de la computación. En el caso de CNNs modernas como Resnet-50 [14] el 98.9 % de las MACs se llevan a cabo en las capas convolucionales. Pero en que consiste una convolución? La figura 2.2 muestra un filtro de dimensiones *R*S*C* que corresponden con la altura, anchura y profundidad del filtro respectivamente (En el caso de la figura S = R). El input fmap tiene unas dimensiones *H*W*C* (por conveniencia se usan filtros de la misma profundidad *C* que el input) y por cada canal(*C*) se realiza una multiplicación de matrices 2-D entre el filtro y la porción de input fmap correspondiente. Cada uno de los resultados de los distintos canales se suman y pasan a formar parte del output fmap, siendo la suma acumulada a lo largo de los C canales un escalar representado en la figura 2.2 por un cuadrado rojo oscuro. Se repite la operación hasta haber multiplicado cada píxel del input fmap por el filtro resultando en una matriz 2-D de dimensiones *P*Q*. Existe un parámetro más, el *stride*, el cual determina el numero de píxeles que se desplaza el filtro en cada barrido. Repitiendo este proceso *M* veces obtenemos un output fmap de las dimensiones *P*Q*M* siendo *M* el numero de filtros.

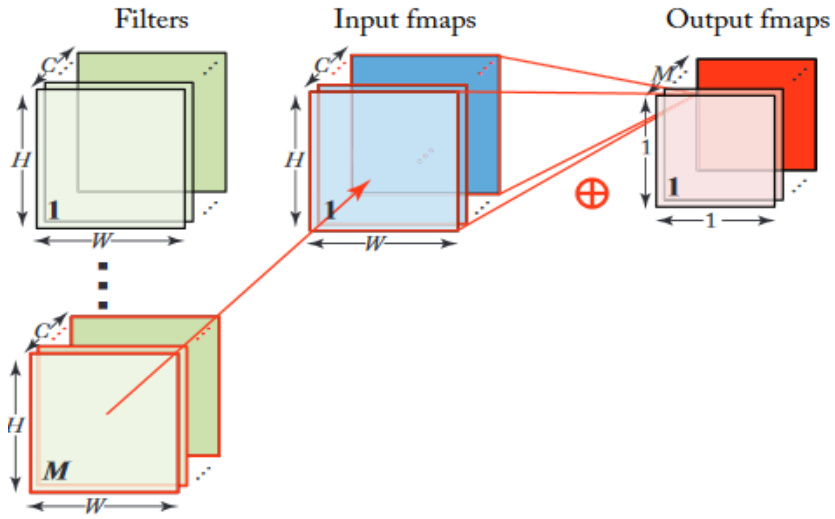

Figura 2.3: Computación llevada a cabo en la FCL. El tamaño del filtro corresponde con el del input feature map. Figura extraída de [12]

FCL: Se puede ver esta capa como una convolución sin compartición de pesos. Siguiendo con la notación anterior, el filtro tenia unas dimensiones *R*S*C* en el caso de la capa convolucional, en cambio en la FCL tiene dimensiones *H*W*C* al igual que el input fmap (ver figura 2.3). Por lo tanto el output fmap es una suma ponderada del input fmap.

Funciones no lineales: Son usadas como funciones de activación para cada resultado de la suma acumulada. En el aprendizaje automático clásico las sigmoides o tangentes hiperbólicas han sido bastante populares. En 2010 [15] introduce ReLu (Rectified linear unit) simplificando de forma significativa la computación de las funciones de activación. En 2013 [16] propone el uso de ReLu para DNNs mostrando mejores resultados que modelos previos que hacían uso de sigmoides. Hoy en día ReLu y sus variantes son una parte esencial de las CNNs ya que además de simplificar el calculo generan *sparsity* en la red, es decir, ayuda a generar 0 en los ouput fmaps, dando lugar a técnicas de optimización como

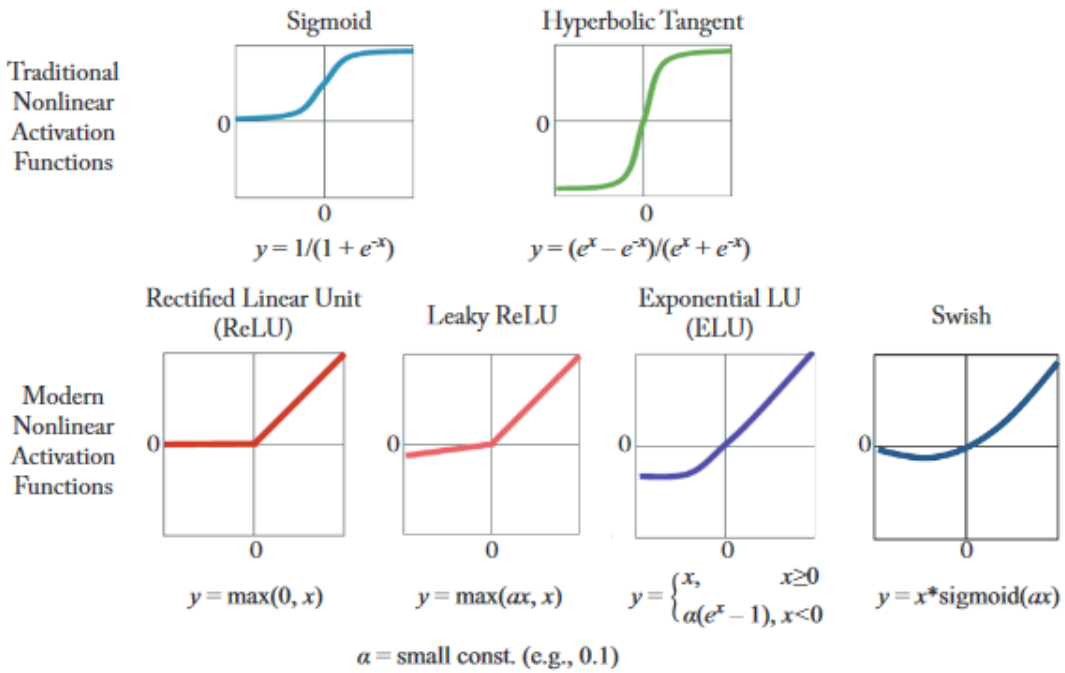

Figura 2.4: Las dos funciones de arriba muestran funciones de activación más tradicionales contra las más modernas ReLU y sus distintas variantes. Su principal ventaja es la simplicidad de calcularlas frente a las sigmoides y la tangente hiperbólica. Figura extraída de [12]

la poda 3.2.1.

Pooling: Esta operación nos permite reducir la resolución de un feature map de la siguiente forma. Al igual que en las capas convolucionales implicaban desplazar una ventana pequeña sobre el el fmap y hacer una convolución, en este caso elegiremos un valor como resultado para dicha ventana o *pool*. Las dos variantes más populares son max pooling y average pooling, donde elegimos el valor más alto del pool o la media de valores del pool respectivamente, ver figura 2.5.

El objetivo del pooling es doble, primero agiliza operaciones posteriores ya que hemos reducido las dimensiones del fmap y segundo es hacer nuestro modelo más robusto frente al ruido. La forma de aplicarlo depende de la arquitectura de la red, pero cuanto mayor sea la pool mayor reducción conseguiremos. Generalmente no superponemos las ventanas, es decir, el stride es igual a las filas de la ventana. Aunque de nuevo, depende de la arquitectura y la aplicación que estemos diseñando, en las CNNs se hace pooling tras la función no lineal que hayamos elegido, siendo otra forma de reducir la dimensión del output fmap junto con la convolución en sí.

2.2. Aceleracion en hardware de RNP

Aunque la computación en la nube este creciendo en popularidad, esta no es apta para muchas aplicaciones en tiempo real. Es el caso por ejemplo de los coches autónomos, en los que la latencia,

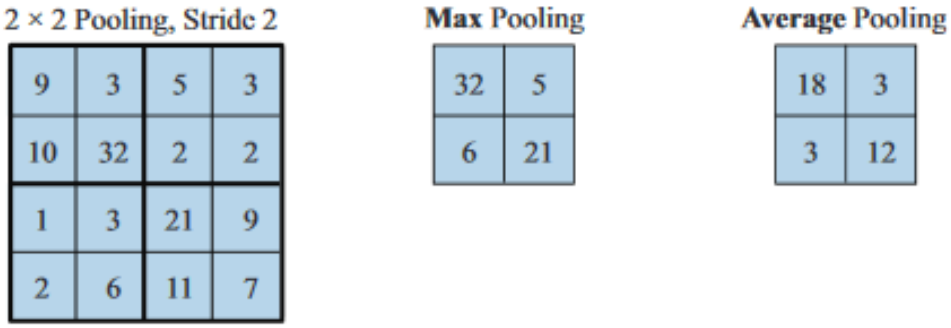

Figura 2.5: Partimos del input feature map de la izquierda y tomamos una ventana de tamaño 2x2 y un stride de 2, de forma que el resultado será 2x2. En el caso del max pooling elegimos el valor máximo dentro de la ventana, resultando en la cuadricula de en medio. Average pooling computa la media de todos los valores, aunque en la practica el calculo extra no merece la pena y se suele utilizar max pooling por su simpleza. Figura extraída de [12]

el consumo de energía y la posible falta de conexión son problemas que dejan fuera a soluciones en la nube o procesadores de propósito general. Además el aumento en complejidad de las CNNs no esta a la par con la mejora en la tecnología de los transistores, lo que lleva a investigadores a tomar nuevos caminos a la hora de abordar los problemas anteriores. La capacidad de poder entrenar redes más grandes ha sido durante mucho tiempo el cuello de botella, un problema que se ha ido solucionando con el uso de multiples GPUs, las cuales mejorar año a año. Aunque existen limitaciones en el paralelismo desde el punto de vista algorítmico, las GPUs son útiles ya que permiten usar un mayor tamaño de lote. En el campo de la inferencia, donde es más común que los lotes sean de una unidad la utilidad de las GPUs queda relegada a su rapidez a la hora de hacer multiplicaciones y sumas matriciales en coma flotante.

Los objetivos del uso de hardware especifico para tareas de inferencia en el Edge son los siguientes, primero ser precisos con los resultados, esta parte parece obvia, pero hay que tener en cuenta que muchos de estos chips se implementan en sistemas críticos y existe muy poco margen de error. Si seguimos con el ejemplo del coche autónomo, el rendimiento (throughput) es esencial, ya que dependiendo de la velocidad del vehículo necesitamos un ratio de predicciones por segundo que para ser considerado tiempo real debería de llegar a las 30 inferencias por segundo. Como es de esperar, la latencia es también un factor clave en sistemas críticos y una de las principales ventajas que nos da la computación en el Edge. Encima de todo esto, el sistema que decidamos implementar tiene que cumplir unos requisitos de consumo de energía que pueden llegar a las unidades de vatio y tener un coste por unidad razonable. Para acabar también seria ideal tener cierta flexibilidad dados los constantes avances en el campo de las DNNs.

2.2.1. Limitaciones de GPUs y CPUs

CPUs y GPUs atienden al mismo principio de localidad temporal, es decir, las ALUs reusan datos y recursos que hayan sido recientemente utilizados. Las ALUs no tienen forma de comunicarse directamente entre ellas, únicamente con la jerarquía de memoria y la forma que tienen de implementar paralelismo es ejecutando múltiples datos en las distintas ALUs o usando paralelismo a nivel de hilo.

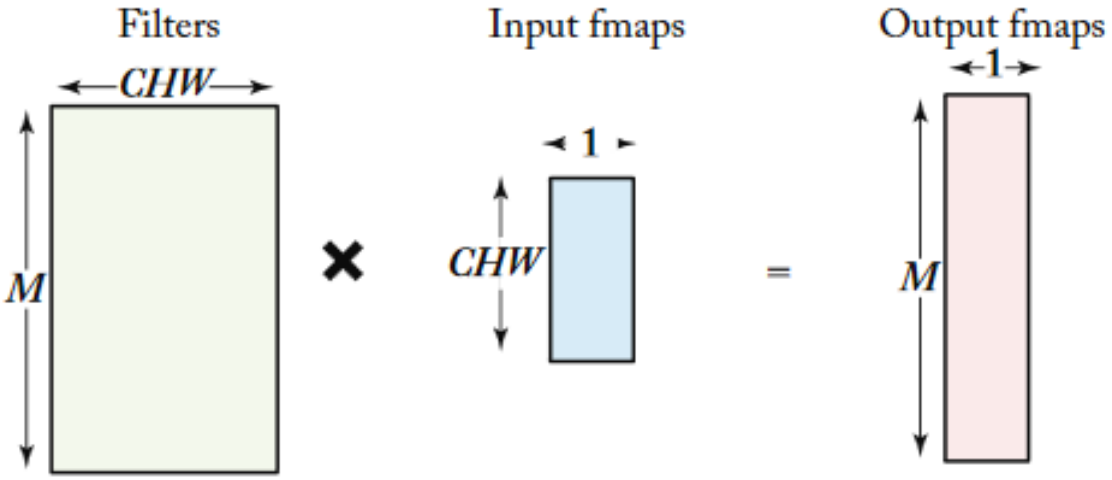

Figura 2.6: Al compartir las dimensiones *CHW* no es necesaria ninguna transformación del input fmap. En este caso multiplicamos una matriz (filtros) por un vector (input fmap) obteniendo un vector de dimensiones el numero de filtros *M*. Figura extraída de [12]

GPUs y CPUs son especialmente buenas en la multiplicación de matrices con operandos en coma flotante, teniendo en cuenta las numerosas librerías que existen para ello (cuDNN por parte de Nvdia por ejemplo). Especialmente las GPUs ya que por norma general ofrecen más unidades de procesamiento que una CPU por el mismo precio. Si recordamos la arquitectura de una CNN tenemos 2 tipos de capas principales y ambas se pueden mapear como una multiplicación matricial con ciertos costes añadidos. El mapeo más simple es para la FCL, ya que tenemos *M* filtros de tamaño *C x H x W* y un input feature map de las mismas dimensiones. El vector output feature map tendrá las dimensiones *1 x M*, ver figura 2.6.

Para las capas convolucionales es algo más complicado, ya que el tamaño de los filtros es menor al del input fmap, requiere usar matrices de Toeplitz, que extiende la matriz del input fmap de forma que se pueda multiplicar por el filtro, como se ve en la figura 2.7. Las librerías de DNNs para CPUs y GPUs se hacen cargo de optimizar estos procesos cuando compilamos nuestros modelos, teniendo en cuenta la memoria on-chip, off-chip y el numero de ALUs del que dispongamos.

Todo esto trae consigo ciertos inconvenientes, el primero, la eficiencia energética, una GPU tiene un consumo en el orden de las centenas de vatio, muy lejos de las unidades de vatio que requieren muchos sistemas. Otro inconveniente para aplicaciones en el edge seria la latencia, puede inducir a error pensar que el alto troughput de las GPUs implica menor latencia, pero este es debido a la

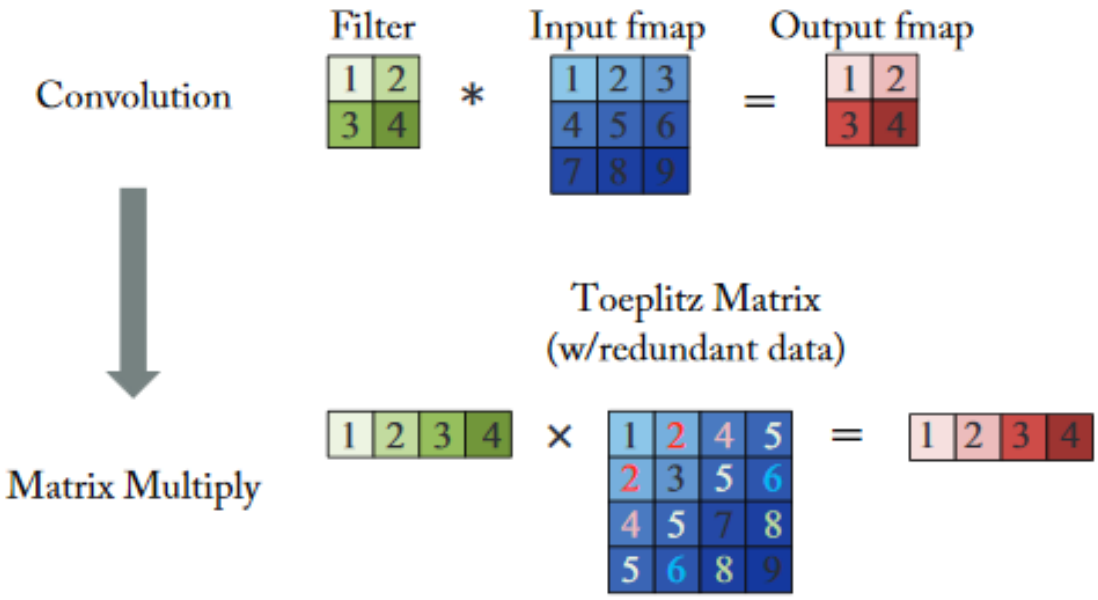

Figura 2.7: Para poder realizar convoluciones como multiplicaciones de matrices se transforma el filtro en un vector, esto requiere expandir los valores de la matriz input fmap de forma que el valor del output fmap no se vea afectado por la transformación. Figura extraída de [12]

capacidad de procesamiento en lotes. El tiempo que pasa desde que recibimos la ultima imagen del lote hasta que la procesamos es pequeño, pero cuanto mayor sea el lote, el tiempo en procesar la primera imagen puede no cumplir los requisitos de latencia. Por ultimo hay que tener en cuenta los requisitos de flexibilidad, como discutiremos en el capitulo 3.2.2, reducir la precisión de los pesos es una técnica muy común, debido al coste de hacer multiplicaciones en coma flotante contra hacerlas en cualquier otra precisión menor, como 8 o 16 bits. Y aunque se puedan desarrollar GPUs que trabajen en distintas precisiones esto requiere el desarrollo de nuevas arquitecturas y el ritmo al que surgen nuevas técnicas y algoritmos es muy difícil de seguir por los desarrolladores de este tipo de hardware. Es por esto que como veremos en el capitulo siguiente las FPGAs son un candidato ideal para llevar acabo inferencia de CNNs en particular y DNNs en general.

2.2.2. Uso de hardware especifico

Son muchos los que predicen la muerte de la ley de Moore, lo que implica que el margen de optimización de DNNs en el ámbito de las GPUs y CPUs es escaso, pero no hay que perder la esperanza ya que esto deja al panorama del hardware especifico como principal protagonista en la carrera del computo de DNNs siendo nuestra principal apuesta para el progreso de la inteligencia artificial.

En el apartado anterior explorábamos como arquitecturas ya extendidas se pueden utilizar para la optimización de CNNs, estas usaban un acercamiento temporal al principio de localidad. En esta sección veremos como arquitecturas que explotan la localidad espacial han demostrado ser muy útiles

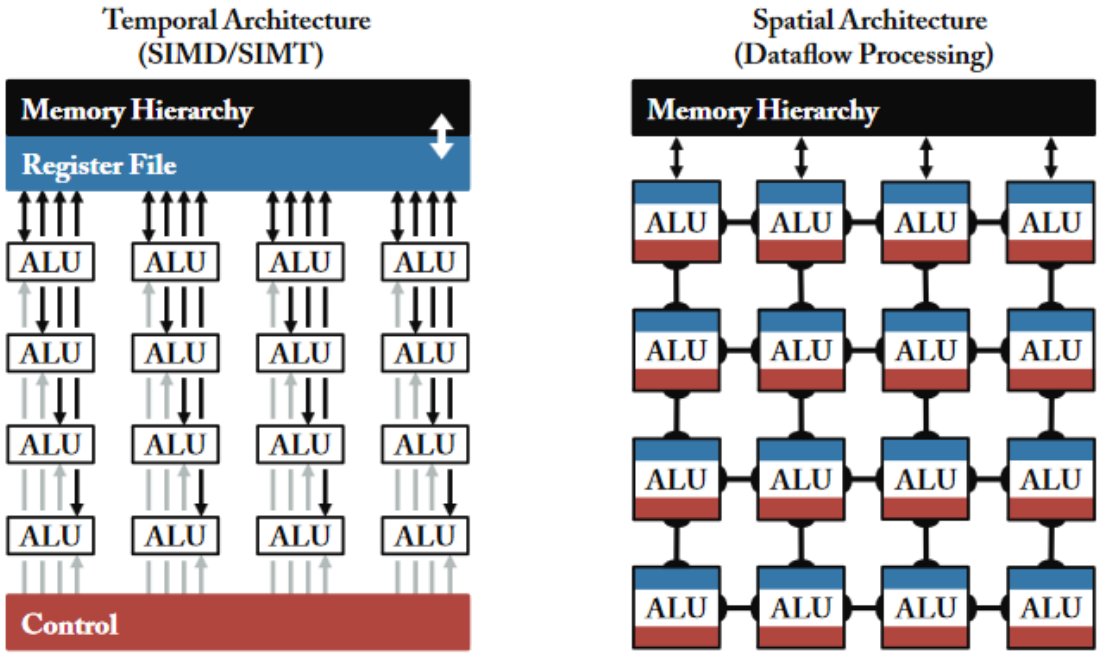

Figura 2.8: Comparación visual de los dos principales acercamientos al principio de localidad. En el caso de la localidad espacial cada una de las ALUs dispone de un fichero de registros y control propios. Figura extraída de [12]

en la computación en el edge. Esto se debe principalmente a la descentralización de la memoria, no solo disminuyendo el coste energético de cada acceso sino que además hace mas eficiente el paralelismo en el caso de las capas convolucionales, ya que los pesos, por ejemplo, se pueden guardar localmente.

Como nos ayuda el hardware especifico entonces a mejorar aquellas métricas en las que los microprocesadores flaqueaban? El primer desafió es mantener un alto troughput sin aumentar la latencia cuando procesamos lotes de tamaño 1. Teniendo en cuenta que el numero de multiplicaciones y acumulaciones (multiply and accumulate, MAC) viene dictado por el modelo que vayamos a ejecutar, debemos maximizar el numero de MACs que podemos hacer en paralelo, no solo aumentando los elementos de procesamiento (EP) en nuestro chip, sino también minimizando los EPs ociosos.

Para reducir el coste energético de la inferencia es necesario conocer que operaciones tienen un mayor consumo, en la figura 2.9 se puede observar que el acceso a DRAM en varios ordenes de magnitud la operación mas costosa comparado con acceso a SRAM o incluso una multiplicación en coma flotante. Para reducir el impacto en el movimiento de datos existen dos técnicas, reducir los accesos a memoria externa y reducir el tamaño de los datos disminuyendo su precisión.

Como ya hemos mencionado anteriormente se destinan muchos recursos al diseño de procesadores para tareas especificas dentro de las DNNs, tanto para entrenamiento como para inferencia. Pero dado el ritmo al que avanza la algoritmia y las técnicas de optimización, su uso no es siempre

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Multiply	0.2
32b Multiply	3.1
16b FP Multiply	1.1
32b FP Multiply	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640

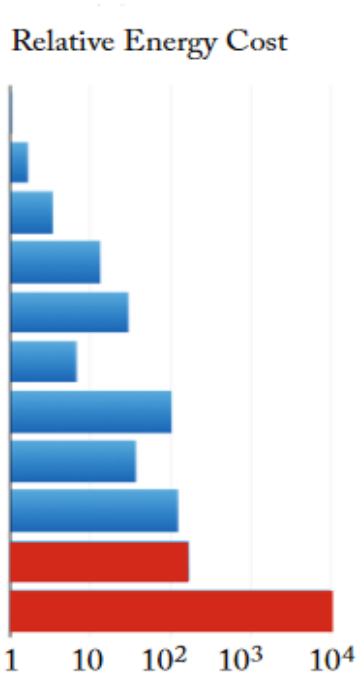

Figura 2.9: Consumo de distintas operaciones con tecnología CMOS de 45mm. El coste se muestra relativo a la suma de 8 bits en escala logarítmica. En azul se muestran operaciones aritméticas contra movimiento de datos en rojo. Conforme aumentamos la precisión de los operandos aumenta el consumo, aunque este es mucho mayor cuando hablamos de lecturas a memoria. Figura extraída de [12]

viable, es por esto que se ha extendido el uso de FPGAs ya que permiten a hardware y software evolucionar acordemente. Desde la década de los 90 que se vienen usando FPGAs para inferencia [17] y en la actualidad compañías como Intel con openVINO o Xilinx con la ya mencionada VitisAI ofrecen herramientas software y soluciones hardware que amplían el mercado de la aceleración en hardware expandiendo enormemente el uso de DNNs a nuevos sectores.

3

Planteamiento

El objetivo de este capitulo es comprender los problemas a los que nos enfrentamos con el co-diseño de hardware y redes neuronales. Se plantea el problema desde dos puntos de vista, el de la visión artificial y el diseño de hardware. Se presenta el algoritmo clave de este trabajo *YOLO* y seguido se exploran diversas técnicas de paralelización, reutilización de datos y optimización de recursos.

3.1. Vision artificial y redes convolucionales

Visión artificial o visión por computador engloba muchos conceptos y problemas, desde estudiar sensores de cámaras a algoritmos tan sofisticados como YOLO en el que nos centraremos mas adelante. En este apartado se explican problemas clásicos de la visión artificial en relación a los algoritmos que se ven normalmente desplegados en sistemas empotrados.

De menor a mayor dificultad podemos distinguir tres tipos de problemas, todos ellos tienen en común el uso satisfactorio de redes neuronales profundas para atajarlos.

Clasificación de Imágenes: Consiste en encontrar elementos dentro de una imagen y listarlos. Puede servir para dividir en temáticas las imágenes, pero su uso en aplicaciones en tiempo real es limitado ya que no localiza los objetos identificados. A principios de década su utilidad principal recaía en comparar distintos algoritmos, y es en la competición de clasificación de imágenes ImageNet [8] de 2012, donde se da el pistoletazo de salida a las DNNs con AlexNet [9].

Detección de Objetos: Con la mejora en la capacidad de computo las DNN han podido escalar a problemas mas complicados como puede ser la detección de objetos. No solo la capacidad de computo ha sido un problema, la dificultad de anotar un dataset de detección es mucho mayor ya que requiere definir bounding boxes (cajas delimitadoras) para cada ítem además de la clase a la que pertenece. En los últimos años han surgido datasets como COCO [19] o PASCAL VOC [20] que han ayudado a progresar en este problema enormemente.

En la actualidad podemos esperar una precisión media de entre el 50/60 % usando modelos basados en CNNs como ResNet [2], SSD [21] o YOLO [22]. Estos avances hacen posible usar estos

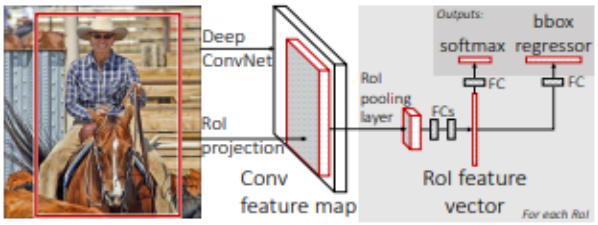

Figura 3.1: Arquitectura de una Fast R-CNN. El input de la CNN consiste de la imagen y las regiones de interes(RoI). Se usa pooling sobre cada RoI y a continuacion una FCL mapea esa RoI a un RoI feature vector. El output de la red consiste de las probabilidades de cada clase (usando softmax) y las bounding boxes. Figura extraída de [18]

algoritmos en tareas que requieren detecciones en tiempo real como la conducción autónoma o robots asistenciales.

Segmentación semántica: La segmentación de por sí consiste en analizar una imagen y determinar que píxeles pertenecen a que estructura/objeto usando distintos métodos de procesamiento de imágenes como watersheds [24] o basados en histogramas [25]. La segmentación semántica no solo realiza la tarea anterior sino que además identifica a que clase pertenecen cada uno de los píxeles de la imagen. Así como la detección dibuja bounding boxes alrededor de los objetos, ahora indicamos las aristas del polígono que contiene el objeto de la imagen. Esto la hace muy útil para problemas de navegación y planificación de robots por ejemplo.

Es fácil ver que esta tarea demanda muchos recursos y generar datasets es también muy costoso por la dificultad de generar anotaciones. Técnicas como el active learning [26, 27] ayudan a acelerar el proceso de aprendizaje y anotación. A pesar de esto datasets populares como CityScapes [28] no contienen mas de 25.000 imágenes anotadas.

Existen numerosos acercamientos con respecto a la segmentación semántica [29], y recientemente el DL con implementaciones como DeepLabv3 [23] usan Deep CNNs consiguiendo muy buenos resultados en cuanto a precisión [29]. Pero cuando hablamos de uso en tiempo real (25/30 fps) las redes mas rápidas como FCN-8 [30] necesitan alrededor de 200ms para realizar una inferencia.

3.1.1. Funcionamiento YOLO

El momento en el que se presenta YOLO [22] en 2016 el rendimiento en tareas de detección de objetos estaba relativamente estancado. La mayoría de métodos se basaban en aplicar clasificación de manera iterativa sobre pequeñas porciones de la imagen, lo cual es terriblemente costoso. Este algoritmo se sofistica y resulta en R-CCN [31], que consiste en proponer regiones de la imagen donde aplicar esta ventana deslizante en forma de convolución en lugar de tener que correr una clasificación completa por cada porción de imagen. Esfuerzos posteriores buscan mejorar la proposición de regiones, y nacen modelos como Fast R-CNN [18].

Figura 3.2: Resultados de aplicar el algoritmo de segmentación semántica DeepLabv3 sobre distintas imágenes. Se consigue distinguir distintas figuras entre ellas y del fondo (en negro). Figura extraída de [23]

YOLO convierte la detección en un único problema de regresión, utilizando una CNN y con un post-procesado muy simple basado en descartar resultados malos o redundantes. De aquí el nombre de YOLO (You Only Look Once). Consiguiendo en su primera versión más del doble de la precisión de otros detectores con el mismo tiempo de inferencia. Evaluar la imagen una única vez no solo trae ventajas de velocidad, algoritmos previos requerían entrenar cada parte del pipeline de manera independiente, YOLO consiste de una única CNN y solo se entrena esta. Además en lugar de ver la imagen como regiones independientes, se codifica la información contextual de la imagen en los filtros de la CNN ayudando a generalizar sobre las representaciones de los objetos.

A continuación se explica el funcionamiento de este algoritmo y como se consigue unificar el problema de la detección. En primer lugar se reajusta el tamaño de la imagen con el tamaño de la primera capa (asumimos 416x416), de forma que el input de la CNN es un tensor [416,416,3], volviendo a la notación de la sección 2.1.1 tenemos *H*W*C* respectivamente. Se divide el input en celdillas de tamaño *S x S* y para cada una de estas celdillas vamos a realizar *B* predicciones, correspondientes al numero de *anchor boxes* que utilicemos, las cuales tienen un tamaño fijo, que se suele determinar usando *k-means*.

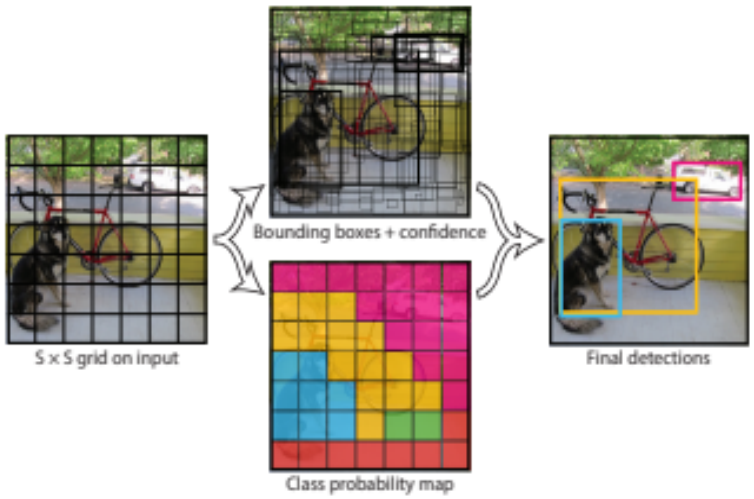

Figura 3.3: YOLO divide la imagen en *S x S* celdillas. Siendo *B* el numero de anchor boxes y *C* el numero de clases a detectar. Existe por cada celdilla un tensor de la siguiente forma *(B * (5 + C))*. Tras aplicar NMS obtenemos las detecciones finales. Figura extraída de [22]

Las predicciones para cada celdilla se codifican de la siguiente forma. Siendo *C* el numero de clases que queramos detectar (En nuestro caso 80 al ser las que contiene COCO dataset), el tensor correspondiente a una celdilla tiene las dimensiones *(B * (5 + C))*. Es decir, se genera una predicción *(5 + C)* por cada *anchor box*. El 5 corresponde con las coordenadas *x, y, w, h* de dicha *anchor box* normalizadas respecto del tamaño de la celdilla (x e y representan los centros y su valor esta comprendido entre 0 y 1, pero anchura y altura pueden superar el tamaño de la celdilla) y la probabilidad de 0 a 1 de que allí se encuentre un ítem independientemente de su clase. . YOLO produce entonces

el siguiente tensor de output *(S x S x (B x (5 + C)))*, en la figura 3.3 podemos ver el pipeline completo.

Al usar Darknet, el framework desarrollado por los creadores de YOLO, el post-procesado de la imagen se implementa en forma de *yolo layer*. En el caso de este trabajo, esta capa no tiene una traducción directa en TensorFlow 1.X, por lo que requiere implementarse por separado, usando OpenCV [32] a la hora del despliegue. El objetivo del post-procesado es eliminar detecciones malas o redundantes. De forma arbitraria decidimos un limite de confianza (refiriéndose a la confianza de cada *anchor box*, normalmente entorno el 30 y 70 %) y eliminamos detecciones con rangos fuera de nuestra imagen y realizamos así un primer filtrado.

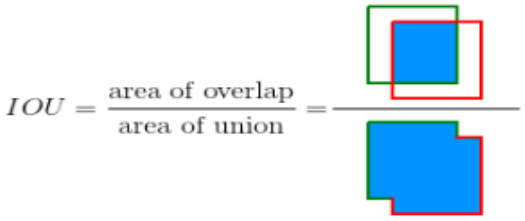

Figura 3.4: El IoU entre dos bounding boxes corresponde a la relación entre su solapamiento y su unión. Dependiendo de la implementación elegiremos un valor limite para determinar si dos bounding boxes han detectado el mismo item, generalmente 0,5.Figura extraída de [33]

El siguiente paso del post-procesado se conoce como NMS (Non-Max-Suppression). Ya que cada celdilla tiene sus propias predicciones es muy probable que un objeto sea detectado por más de una. NMS toma aquella bounding box con la mayor confianza y calcula la IoU (Intersection over Union, ver figura 3.4) y aquellas bounding boxes con confianzas menores y un IoU por encima de un limite (0.5 normalmente, aunque en YOLOv4 es de 0.213) se consideran que pertenecen al mismo objeto y se descartan. Realizamos este proceso para cada clase *C* y esto nos deja con las detecciones más prometedoras. Solo queda darles el formato deseado, que dependerá de si queremos dibujar las detecciones en la imagen o guardarlas para su evaluación.

En esta primera versión de 2016 los creadores apuntan que el algoritmo tiene problemas detectando objetos pequeños. En versiones posteriores esto se ha atajado con el uso de varios tamaños de celdilla distintos, de forma que la red produce primero el output para las celdillas más grandes, 19 x 19 sobre la imagen, y va escalando hasta 78x78 dependiendo de la versión y el objetivo de nuestra aplicación.

3.2. Aceleración de inferencia en FPGAs

A continuación exploraremos ciertas características de las CNNs que podemos aprovechar gracias a la flexibilidad de las FPGAs. En el apartado sobre el estado del arte veíamos brevemente como GPUs y CPUs tenían ciertas limitaciones para paralelizar las capas convolucionales, y como estas se mapea-

ban a multiplicaciones de matrices. Aunque este mapeo siga siendo necesario, los costes añadidos de la conversión se ven reducidos debido a la mayor capacidad de las FPGAs para la reutilización de datos además de muchas otras técnicas que veremos en este capítulo.

Los objetivos del uso de FPGAs siguen siendo los de optimizar al máximo la paralelización que ofrecen las CNNs sin los costes añadidos de tener que diseñar chips desde cero para tareas específicas. Además de aprovecharnos de toda la literatura y software ya existente al respecto, permitiendo a desarrolladores de software sin experiencia en el diseño de FPGAs optimizar sus modelos de aprendizaje automático y multiplicar el rendimiento del despliegue de todo tipo de aplicaciones de visión artificial.

Por ultimo veremos en que consiste VitisAI, un kit de herramientas opensource que pretende acercar el desarrollo de hw y sw para la optimización de inferencia. Con VitisAI podremos poner en practica técnicas de cuantificación y poda, además de compilar nuestra CNN acorde al hardware en el que se hará la inferencia.

3.2.1. Técnicas para explotar CNNs

A pesar de la alta dimensionalidad de las CNNs estas muestran una serie de características que podemos explotar, relacionadas principalmente con la redundancia de datos. Existen numerosas formas de atajar este problema, pero como es obvio prestaremos más atención a aquellas técnicas que tengan un mayor impacto métricas como el consumo energético, la latencia o el throughput. Esto nos lleva inevitablemente a la ya mencionada MAC, operación omnipresente tanto en las capas FC como en las convolucionales.

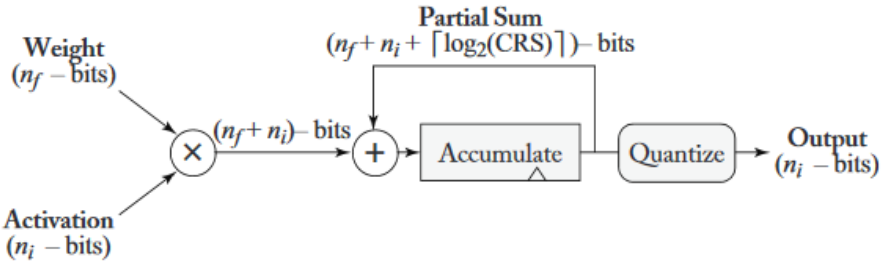

Figura 3.5: Diagrama de una operacion MAC, en el contexto de las CNNs multiplicamos las activaciones (input fmaps) con los pesos (filtros), sumando todos los resultados (paramos al acabar de computar el input fmap correspondiente) obtenemos el output. Debajo de cada uno de los operandos se indica la precisión usada para representarlos, cuantizando el resultado de la suma parcial estamos cuantizando indirectamente las activaciones futuras. Figura extraída de [12]

Cada una de las millones de MACs que nuestros EP tendrán que ejecutar se divide en 4 accesos a memoria, ver figura 3.5. Por una parte tenemos que multiplicar el filtro por la porción del fmap que corresponda (En el caso de una FCL esa porción corresponde al completo del fmap), recuperar la suma parcial, realizar la suma y actualizar el resultado acumulado. Si realizamos del orden de las cientos

de millones de MACs para ejecutar una CNN como ResNet-50, tendríamos 4 veces ese numero de accesos a memoria.

Un acceso a memoria DRAM de 32 bits con tecnología CMOS de 45mm consume 640pJ, mientras que acceder a una memoria más cercana al EP como la SRAM tiene un consumo de 5pJ, la figura 2.9 contiene una comparativa de las distintas operaciones y su consumo. Como referencia [34], ejecutando una red neuronal de 1 billón de conexiones sin ningún tipo de optimización a 20 fps se consumirían 12.8 W solo en accesos a DRAM. Esta es la razón por la que la mayoría de técnicas de optimización giran entorno a reducir la cantidad de accesos a memoria, bien aprovechando el ancho de banda de la memoria al máximo o reduciendo el tamaño del modelo con cuidado de no perder demasiada precisión.

Por suerte la arquitectura inherentemente redundante de las CNNs da lugar a poder reutilizar muchos de los operandos que intervienen en un MAC. En las capas convolucionales vamos a multiplicar un filtro a lo largo del input fmap, por lo que no es necesario recuperarlo en cada MAC. En las FCL y capas convolucionales distintos filtros se aplican sobre la misma región del input fmap, dependiendo del numero de canales del output fmap. Aunque el nivel de reutilización depende en parte de la forma de los input/output fmaps y de los filtros además del valor del stride. Siguiendo el principio de localidad temporal (Figura 2.8), este requería a cada ALU acceder a los datos a través de la jerarquía de memoria, permitiendo guardar los datos que creamos vamos a reutilizar en los nivel inferiores y más baratos (teniendo en cuenta el tamaño de unos buffers que son relativamente pequeños). Se puede aprovechar el principio de localidad espacial de forma que acerquemos al máximo los datos al lugar donde se realiza la computación, permitiendo a cada uno de los EP intercambiar datos entre ellos, intentando minimizar los accesos a niveles de memoria más costosos.

Existe una gran dependencia entre la arquitectura de la CNN y la arquitectura del hardware a la hora de encontrar una configuración óptima, llamamos dataflows a las decisiones de diseño que se preocupan de optimizar el uso de la memoria en estos casos [13]. De esta forma ajustamos diferentes parámetros hardware como la memoria de cada EP, o el ancho de banda de la memoria al modelo. La memoria de cada EP es limitada lo que añade dificultad a la hora de diseñar dataflows. Si añadimos a esto la tendencia en el aumento de complejidad de los sistemas de visión artificial basados en CNNs tenemos un espacio de diseño enorme en el que es complicado encontrar la solución óptima a nuestro problema. Existen diferentes acercamientos descritos [13] en en función de que tipo de datos elegimos guardar de forma local y como los distribuimos a lo largo de la jerarquía de memoria, la figura 3.6 muestra un resumen de las diferentes opciones.

Weight Stationary Dataflow: Guardamos dentro del fichero de registro (FR) uno de los pesos del filtro que este en utilización actualmente. Idealmente usaremos ese peso tantas veces como píxeles haya en el output fmap (E^2) multiplicado a su vez por el numero de canales del input fmap (N), es decir E^2N usos de un mismo peso una vez lo hemos recuperado de DRAM.

Esto requiere transmitir cada uno de los píxeles del input fmap a los EPs que contienen los pesos en

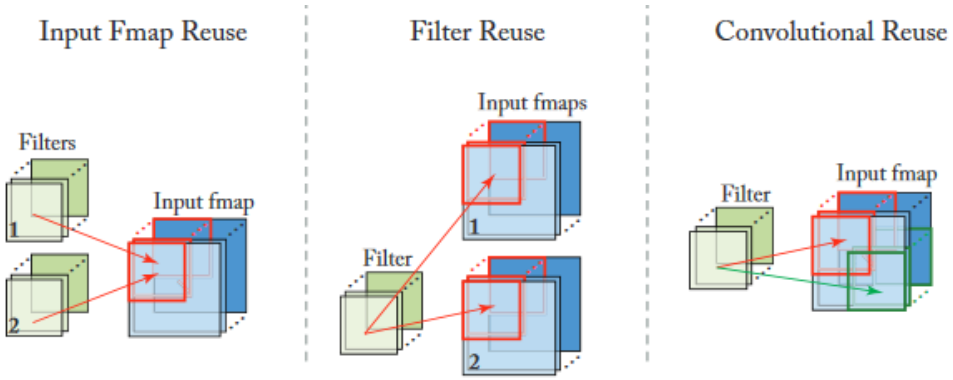

Figura 3.6: Distintos acercamientos en cuanto a reutilización de datos. De izquierda a derecha, reutilización del input fmap, pesos y resultados parciales de las convoluciones. El objetivo es minimizar el movimiento de datos para reducir los accesos a memoria, reduciendo así el consumo. Figura extraída de [12]

su FR, además se usa el FR para guardar las sumas parciales las cuales se comparten entre EPs y se escriben de nuevo en el buffer global, y dependiendo del tamaño del buffer global la porción de la suma parcial que podamos almacenar localmente limitara el porcentaje de pesos que podamos almacenar on-chip. Hay que tener en cuenta que un FR suele contar con 1KB de memoria, lo que hace inevitable tener que acceder a memoria externa más costosa, en este diseño y en los siguientes.

Ouput Stationary Dataflow: En este caso el FR contiene los valores del output fmap además de las sumas parciales, esto reduce lecturas como el método anterior pero además las escrituras, y se reutiliza la suma parcial. Los pesos se envían a cada EP desde el buffer global al realizar una nueva MAC. Cada EP tiene que acceder a un EP vecino para obtener las activaciones previas, siendo estas más eficientes que tener que acceder al buffer global.

Input Stationary Dataflow: Al comienzo del procesamiento de cada capa se distribuyen los pixeles del input a lo largo del array de EPs, estos se reusan y en cada MAC se distribuyen los pesos. Los EPs comparten cada una de las sumas parciales para al final del computo actualizar este resultado en el buffer global.

Row Stationary Dataflow: Este método [13] es algo más complejo que los métodos anteriores ya que intenta reutilizar todos los operandos de la MAC. Implica partir una convolución multidimensional en varias operaciones vectoriales y guardar los operandos de cada una en un EP. Si se realiza correctamente se estarían reutilizando los pesos y el input fmap del FR. Su correcto funcionamiento depende del mapeo de estas convoluciones 1D al array de EPs, lo cual no es trivial.

Hasta aquí nos hemos enfocado en reducir el consumo, y aunque esto también tenga un impacto en el throughput indirectamente en esta sección veremos técnicas de paralelización que van dirigidas a mejorar esta ultima métrica. [35] Define cuatro niveles a los cuales se puede paralelizar una CNN.

Paralelismo a nivel de tarea: Si vamos a hacer inferencia sobre varias imágenes cada una de

las inferencias es independiente de la anterior. Si dispusiéramos de suficiente de ancho de banda de memoria podríamos explotar esto, además de poder reutilizar los pesos entre inferencias reduciendo también los accesos a memoria. Este acercamiento es del que se aprovechan GPUs ya que suelen disponer de más capacidad de memoria que las FPGAs, pero como ya hemos visto anteriormente, no nos interesa tanto en este trabajo mejorar el rendimiento del procesamiento por lotes.

Paralelismo a nivel de capa: El paralelismo entre capas es muy limitado ya que hay una dependencia entre una capa y la siguiente. Pero parecido al método anterior podemos mejorar el rendimiento si hacemos un pipeline entre las capas de una misma inferencia permitiendo ejecutar en paralelo las capas de la siguiente imagen.

Paralelismo a nivel de bucle: Dentro de una capa convolucional cada convolución es independiente, característica que también explotábamos en los dataflows para poder reutilizar datos. En este caso consideramos que dentro de un bucle se esta ejecutando estas convoluciones independientes entre sí, lo que implica que técnicas como el desenrollado de bucles (loop-unrolling) seria aplicable, únicamente limitada por la capacidad de memoria on-chip.

Paralelismo a nivel de MAC: Recordamos que la multiplicación filtro x input fmap se mapea a varias multiplicaciones vectoriales, cada una conteniendo varias MACs, todas ellas independientes entre si. De nuevo podemos ejecutar cada una en paralelo limitados por la capacidad de memoria on-chip. Si creamos una estructura de árbol dentro de un EP o entre EPs de forma que ejecute las sumas parciales podemos aprovechar la independencia de MACs.

3.2.2. Cuantificación

El uso eficiente de la memoria es clave tanto para obtener un consumo de energía bajo como para el rendimiento y la latencia. Las técnicas que hemos visto ahora se basaban en aprovechar el paralelismo que permiten las CNNs y la reutilización de datos, pero en este apartado veremos como reducir la precisión de las activaciones, pesos o sumas parciales puede impactar positivamente en todas las métricas.

La idea es muy intuitiva, si reducimos a la mitad la longitud de nuestros operandos podemos doblar el numero de operandos en memoria on-chip y doblar el rendimiento de los accesos a memoria. Podemos reducir a la mitad el tamaño de los FR de los EP, reduciendo el coste de cada EP lo que impactaría directamente en nuestro rendimiento MACs/ciclo. O alternativamente almacenar el doble de operandos como mencionábamos antes. En la sección anterior mencionábamos como una lectura de 32b a memoria era la operación más costosa en relación al resto, si podemos doblar el rendimiento de las lecturas reduciríamos enormemente su impacto energético y temporal.

Además de ayudar a optimizar el uso de memoria, las operaciones escalan en complejidad conforme aumentamos la precisión de los operandos. En concreto el espacio y energía de realizar una

Figura 3.7: Diferentes precisiones y tamaños de representación. *S* representa el bit de signo, n_m bits sirvenpara representar 2^{n_m} numeros en el caso de los enteros. Representaciones en FP usan ademas *E* bits para representar exponentes. Figura extraída de [12]

multiplicación es $O(n^2)$ siendo n el numero de bits de los operandos. El retardo en el camino crítico de la multiplicación y suma escalan linearmente dependiendo de la implementación. La figura 3.5 muestra como el resultado de multiplicar dos operandos de distinta longitud n_f y n_i tendrá longitud igual a la suma de ambos $n_f + n_i$. Tras computar todas las sumas parciales se reduce la precisión de la acumulación final a la misma precisión n_i que servirá de input para la siguiente capa.

Model	Top-1	Top-5
Binarized activations+weights, during training and test		
BNN	47.1%	69.1%
Quantize weights and activations during training and test		
QNN 4-bit	66.5%	83.4%
Quantize activation,weights and gradients during training and test		
QNN 6-bit	66.4%	83.1%
No Quantization (standard results)		
GoogleNet - our implementation	71.6%	91.2%

Figura 3.8: Distintas implementaciones de GoogleNet y como reducir la precisión impacta en el Top-1 y Top-5 accuracy. Si comparamos con el modelo estándar, reducirla precisión de todos los operandos a 4 y 6 bits no tiene un impacto demasiado notable, apenas un 5 % en Top-1 accuracy. Figura extraída de [36]

Ahora la pregunta es, que precisión elijo y que efectos negativos esta puede tener en la exactitud de mi modelo. [37] muestra como existe una gran redundancia en los pesos y como las CNNs están sobre-parametrizadas. En el momento del entrenamiento es útil ya que evita el sobreajuste de una red tan compleja, pero a la hora de llevar a cabo la inferencia un gran porcentaje de los pesos no aportan ningún valor a la predicción. Gracias a esto podemos reducir el numero de bits para representar operandos sin apenas perdidas en la precisión del modelo, la figura 3.8 muestra diferentes cuantificaciones

de GoogleNet [36] y su impacto en Top-1 y Top-5 accuracy.

Existen numerosas configuraciones, pero en este trabajo hemos optado por pesos y activaciones con enteros de 8 bits, la figura 3.7 muestra diferentes de representaciones numéricas. Las sumas parciales siguen con la precisión de 32 bits en coma flotante pero se truncan a la anchura de las activaciones una vez se ha acabado el computo [35]. Con esta configuración la perdida de precisión es mínima y conseguimos reducir x4 el tamaño del modelo, además de la mejora en rendimiento y consumo de energía que discutiremos en el apartado 4.3 sobre resultados.

Los esfuerzos por reducir aun más la anchura en bits de los operandos y mantener la precisión continúan y trabajos como [38] dan buenos resultados llegando a usar un único bit de representación. Otros esfuerzos se focalizan en realizar ya el entrenamiento con pesos truncados [39] con resultados también prometedores.

3.2.3. Poda

Hasta ahora se han propuesto técnicas centradas en optimizar los costes de ejecutar el modelo sin apenas modificar su arquitectura, solo su implementación. La poda se aprovecha de la sobre-paremetrizacion de la que hablábamos en el apartado anterior de una forma más directa, eliminando conexiones redundantes o que aportan poca información. Esto reduce considerablemente el numero de MACs y el tamaño del modelo hasta en un 90 % [40] sin perdidas notables en la precisión, como muestra la figura 3.9.

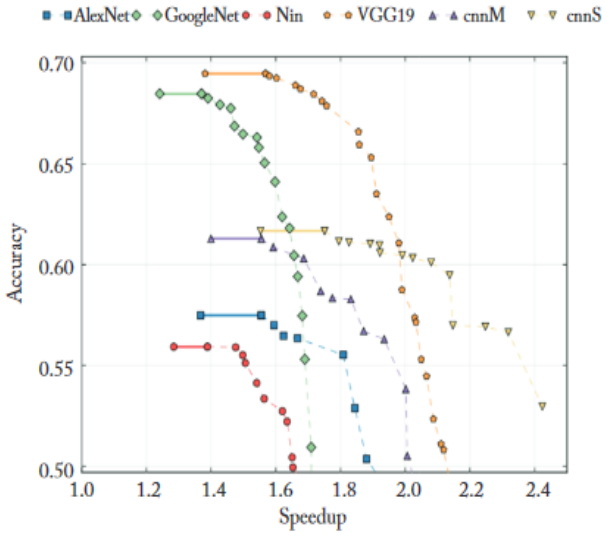

Figura 3.9: Relación entre la aceleración conseguida con la poda y la precisión. Cada uno de los puntos de la curva representa un modelo con distintos grados de poda. Al prescindir de más datos aumenta la velocidad pero disminuye con ella la precisión. Figura extraída de [12]

La poda es una tarea compleja, a continuación se presentan 3 acercamientos con distintos niveles de sofisticación y objetivos.

Poda de pesos basada en magnitud: El objetivo es mantener la precisión de nuestro modelo en un cierto margen reduciendo al máximo el numero de pesos. La pregunta entonces es de que pesos podemos prescindir. La respuesta yace en las operaciones que realizamos en cada capa, MACs además del computo de la ReLu, la primera operación nos permitiría prescindir de aquellos pesos iguales a 0, ya que su multiplicación será 0 y no tendrá ningún efecto en la suma acumulada y por consiguiente tampoco en el output fmap.

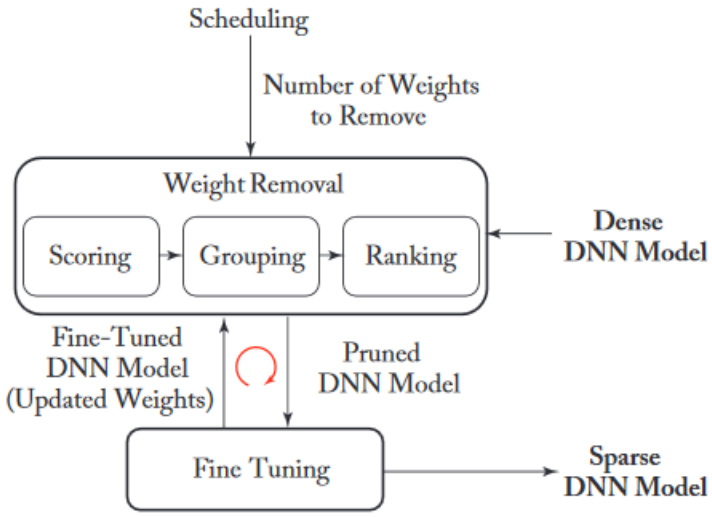

Figura 3.10: Flowchart para podar un modelo. Se parte de un modelo denso y se determina de que pesos prescindir, se reajusta y por ultimo se determina si podemos repetir el proceso o la perdida de precisión no es admisible. Figura extraída de [12]

Para poner esto en practica [34] recorta las conexiones con pesos de menos valor y reajustan los pesos restantes de forma iterativa con propagación hacia atrás, así hasta que se ve afectada la precisión del modelo, como muestra la figura 3.10. Este método es el más extendido y es fácil de implementar en comparación con los que veremos a continuación, además de ser muy eficiente, ver figura X. Con la desventaja de que no tiene en cuenta las dependencias entre pesos lo que limita el ratio de compresión que podemos conseguir en comparación con [41].

Una extensión de podar pesos seria [42], demostrando que es viable eliminar filtros completos. Se prescinde de aquellos filtros que producen un mayor numero de 0 en las activaciones o incluso se llegan a fusionar filtros. Se aplica de forma similar al método anterior, requiriendo poda y reajuste de forma iterativa. Su uso no esta tan extendido como el del método anterior debido a una perdida de precisión más agresiva. El debate esta abierto en cuanto a la *granularidad* de la poda, en la figura 3.11 podemos ver distintos acercamientos, cada uno con sus pros y contras.

Poda Hardware-Aware: Reducir el numero de conexiones y por lo tanto de MACs por norma general nos va a ayudar a mejorar las distintas métricas de nuestro modelo. Pero si queremos tener un impacto más directo en las métricas que mejor forma de hacerlo que tomando estas mismas como referencia.

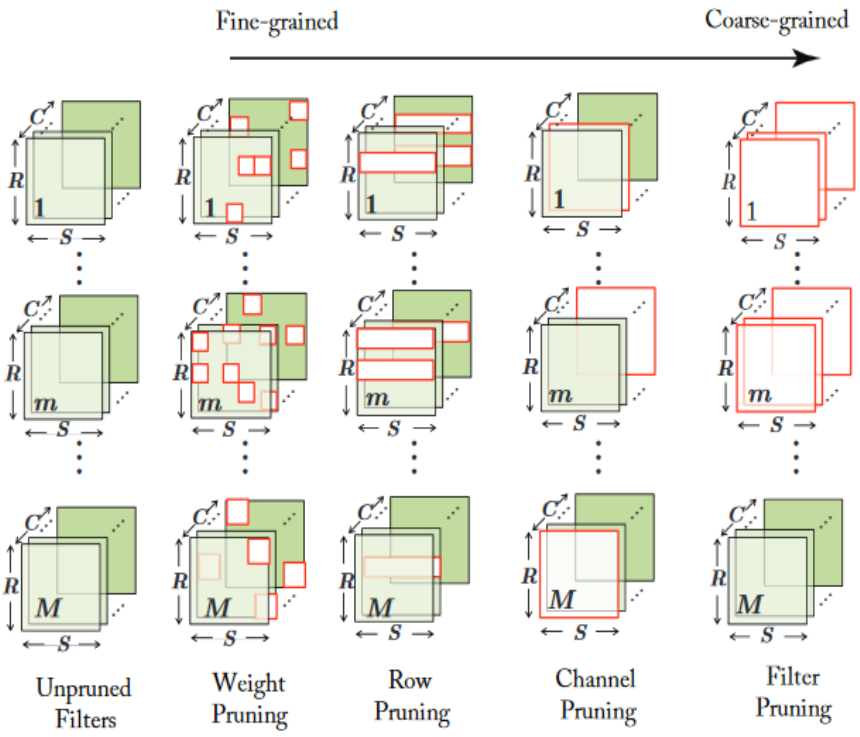

Figura 3.11: La granularidad aumenta de izquierda a derecha. Se comienza podando valores específicos del filtro, posteriormente filas y así hasta llegar a eliminar filtros por completo. Figura extraída de [12]

[41] Busca reducir la energía necesaria para llevara cabo una inferencia, eliminando aquellos pesos responsables de un mayor consumo. La relación pesos/consumo de energía no es lineal, como ya hemos visto, los accesos a memoria son la operación más costosa. Se comienza podando conexiones en las capas que consuman más energía, y se reajustan los pesos dentro de cada capa. Y al igual que en el método anterior, al acabar se realiza un reajuste global de los pesos.

NetAdapt [43] por otro lado se ocupa de reducir la latencia introduciendo un nuevo paso en el proceso de reajuste y poda. Consiste en medir la repercusión que tiene en la latencia podar los pesos de cada filtro, y para obtener resultados óptimos la latencia se debe de medir en el dispositivo sobre el que desplegaremos nuestra CNN. Los desarrolladores de esta herramienta indican una mejora de hasta 1.6x en la latencia sobre MobileNetV1 [44] contra algoritmos de poda más tradicionales.

Uno de los principales desafíos de atacar las métricas es precisamente medirlas, y suele requerir uso de herramientas hardware o software externas. Esto implica que las CNNs resultantes están optimizadas para el dispositivo en concreto sobre el que se hayan realizado las mediciones. La necesidad de co-diseñar algoritmos y hardware se hace evidente si lo que queremos es sacar el máximo partido de nuestro modelo.

3.2.4. Vitis AI

Parte del éxito de las GPUs y procesadores de propósito general se debe a la facilidad de implementar algoritmos de visión artificial y optimizarlos usando librerías como CUDA. Históricamente el desarrollo en FPGAs ha quedado relegado a arquitectos de hardware y profesionales especializados. Con el fin de cerrar esta brecha nace Vitis AI, permitiendo usar frameworks ampliamente extendidos como Caffe o TensorFlow y desplegar DNNs usando herramientas y lenguajes de programación como C++. Vitis AI esta formado por una serie de módulos y herramientas que se explican a continuación.

Una parte importante de Vitis AI es la librería [] opensource que ofrece, junto con numerosas implementaciones de ejemplo. En este trabajo han sido de gran utilidad como referencia hora de implementar YOLOv4 y como benchmark (ver sección 4.3 con los resultados).

Dentro de la parte del kit de desarrollo (ver figura 3.12), hay tres herramientas principales, AI Quantizer, AI Compiler y AI Optimizer. El Quantizer realiza un proceso de cuantización como se expone en la sección 3.2.2 y su uso se explica en detalle en la sección 4.1.2. En principio da soporte a modelos implementados en TensorFlow 1.X o Caffe, aunque dentro de la librería de Vitis AI podemos encontrar conversores dependiendo del framework de origen que hayamos usado. La cuantización transforma los parámetros de nuestra DNN de FP32 bits a INT de 8 bits. Se puede elegir no cuantizar una capa en el caso que contenga una operación no soportada, y la API DPUv2 permite pasar su computo al procesador de propósito general del sistema empotrado que estemos usando.

El Optimizer poda y reajusta el modelo inicial y clama reducir su complejidad entre 5 y 50 veces.

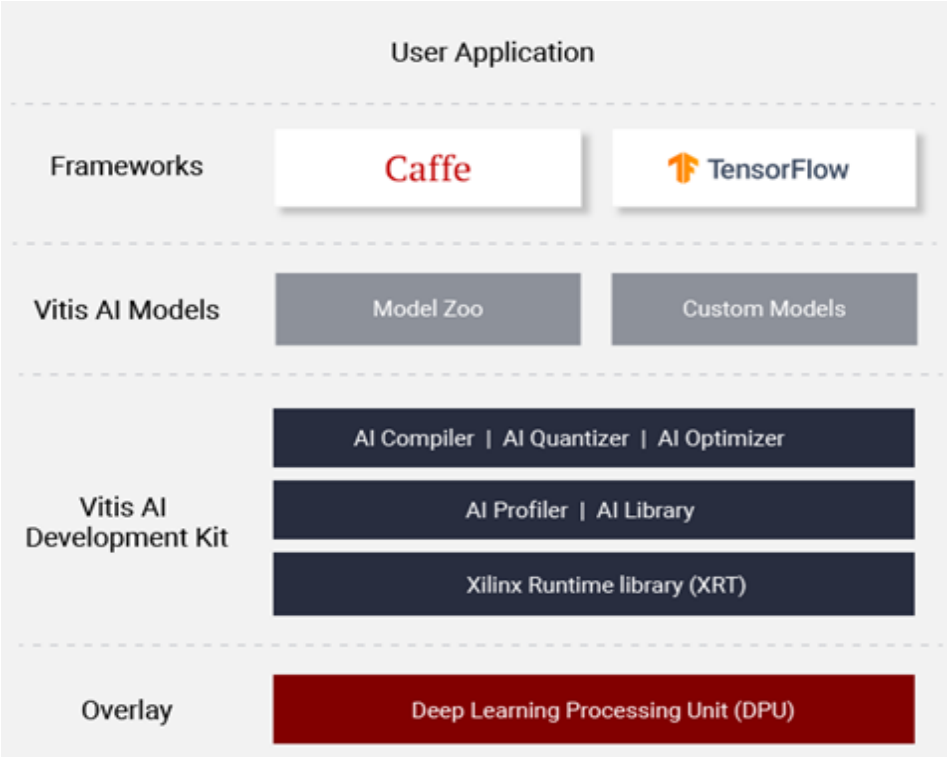

Figura 3.12: Herramientas recogidas en el entorno de Vitis AI, de arriba abajo tenemos la implementación usando frameworks de más alto nivel, hasta el despliegue usando el compilador a DPU. Figura extraída de [45]

Su uso requiere una licencia aunque existen modelos optimizados con esta herramienta en la librería open source.

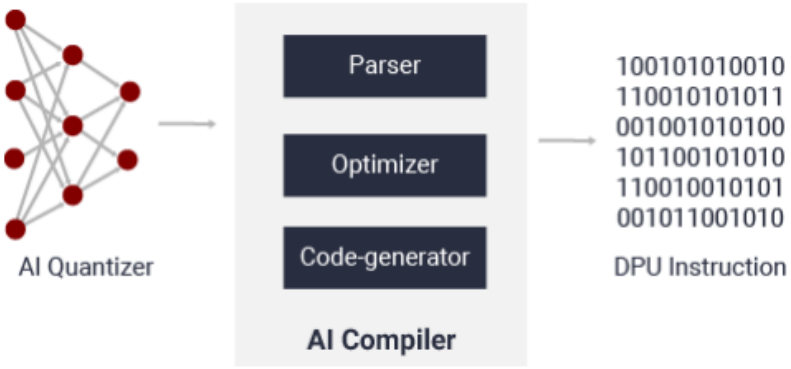

Figura 3.13: El modelo una vez cuantizado puede ser compilado usando AI Compiler. Figura extraída de [45]

Para poder compilar el modelo es necesario haber usado como mínimo el Quantizer, ya que este sirve de input al compilador(ver figura 3.13). El compilador parsea el fichero generado tras la cuantización y lleva a cabo cierta optimización, llegando a reducir el tamaño del modelo, en el caso de este trabajo hasta 4 veces respecto al modelo cuantizado y por ultimo genera las instrucciones para ser ejecutadas directamente por la DPU.

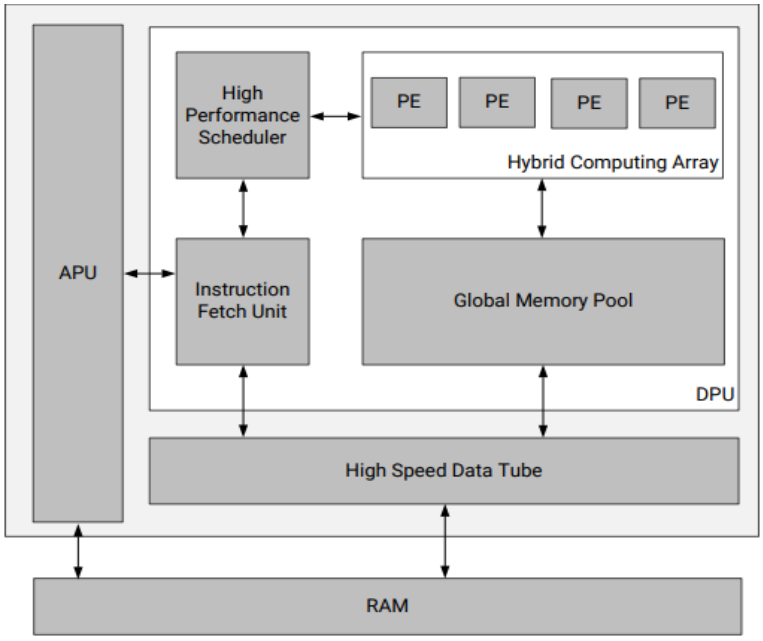

Figura 3.14: Diagrama de alto nivel del bloque que contiene la DPU. APU y PE hacen referencia a Application Processing Unit y Processing Engine (En este trabajo se le ha llamado EP) respectivamente. Figura extraída de [46]

Por ultimo explicar en que consiste la DPU, la figura 3.14 muestra la DPUv2 en conjunto con los recursos más inmediatos. El bloque de la DPU se integra en la lógica programable (PL) de la plataforma

e interactúa con la unidad de procesamiento y con la memoria RAM haciendo uso de interfaces de alto rendimiento. En la figura 3.15 se muestra como se integra la DPU con el resto de los componentes del sistema empotrado, con los cuales tiene comunicación directa mediante la interfaz AXI.

La DPU soporta un gran numero de operaciones como las que se describen en la sección 2.1.1 y dependiendo de la arquitectura soportada por el sistema, en un ciclo de reloj se pueden ejecutar entre 256 y 2048 MACs. Y dependiendo de la capacidad de la PL, se podrían desplegar 4 núcleos de DPU en un sistema simultáneamente.

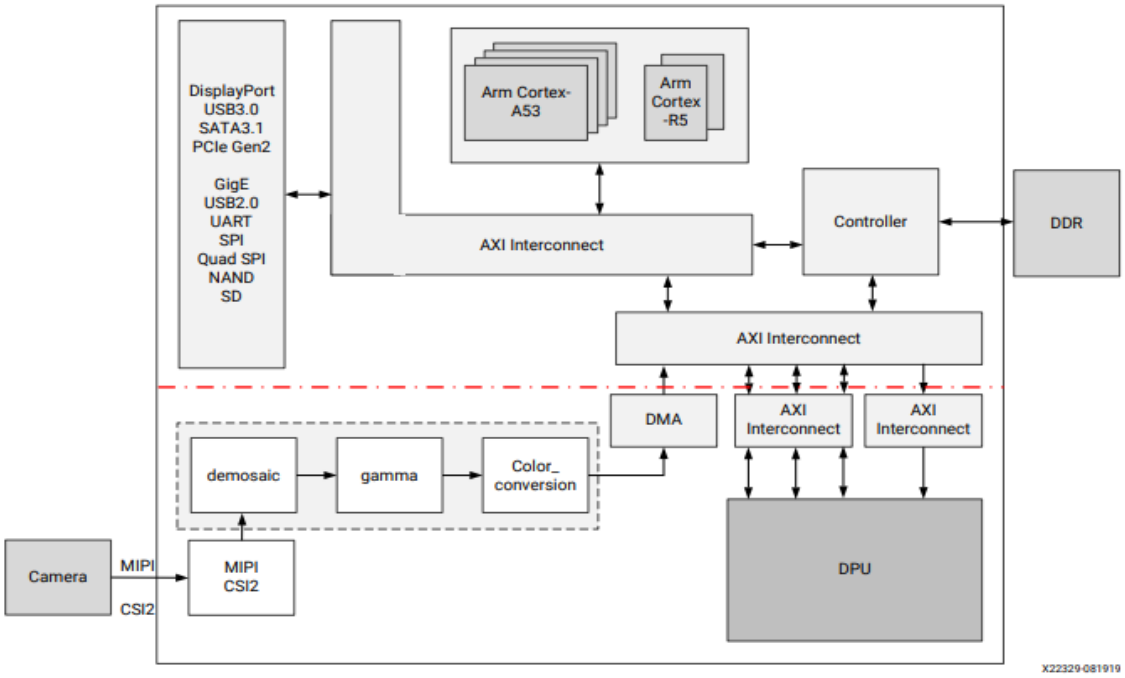

Figura 3.15: Ejemplo de un sistema que integra una DPU. Cabe destacar como el pipeline de vídeo(abajo a la izquierda) completo se hace de forma directa sin necesidad de tomar recursos del procesador principal, en este caso un Arm Cortex-A53, acelerando el procesamiento de vídeo. Figura extraída de [46]

4

Implementación y resultados

A continuación se presentan los resultados del modelo desarrollado en este trabajo en la plataforma ZCU104. El objetivo es comprender como se sitúa el algoritmo YOLOv4 frente a sus competidores en cuanto al despliegue en un sistema empotrado de altas prestaciones. Además se comparara también la plataforma contra otros tipos de sistemas como GPUs, de forma que se pueda medir el impacto de las medidas de optimización explicadas en el capitulo anterior. También se explica cada paso del workflow para desplegar el algoritmo desde su implementación en Darknet.

4.1. Desplegar YOLOv4 en una FPGA

YOLOv3 es uno de los modelos que mejor rendimiento da en su despliegue en DPU, y es uno de los más documentados, existiendo numerosas herramientas para convertir el modelo original en DarkNet a Tensorflow (Hay que tener en cuenta que las herramientas dentro de Vitis AI están hechas para modelos en Tensorflow y Caffe). Con la reciente publicación de YOLOv4 la decisión fue comprobar si su rendimiento una FPGA era tan bueno como aclamaban sus desarrolladores para GPUs. Los cuales mostraban el siguiente gráfico 4.1, donde el nuevo modelo era capaz de mejorar la precisión en más de 10 puntos sin compromisos en el rendimiento en fotogramas por segundo comparado con la versión anterior.

Existen numerosas versiones de YOLOv4, las podemos explorar en [47]. Si queremos usar el compilador de VitisAI nos tenemos que asegurar que la arquitectura de nuestro modelo esta soportada por la DPU. La versión original de YOLOv4 contiene unas funciones de activación relativamente sofisticadas, llamadas *Mish*, que a día de hoy no tienen soporte para DPU. Esto lo podemos solucionar con uno de los modelos que hace uso de funciones de activación *Leaky* (representada en la figura 2.4), el impacto es apenas un punto en la precisión. Si además queremos obtener resultados en tiempo real, conviene reducir el tamaño de la primera capa de la red, de forma predeterminada YOLOv4 acepta imágenes de 608x608, muy por encima de la precisión que usan los modelos desplegados en la ZCU104, que en el caso de YOLOv3 suele ser de 416x416. Teniendo todo esto en cuenta el modelo desplegado es YOLOv4-Leaky-416, siendo el óptimo para tareas en tiempo real ya que es la versión

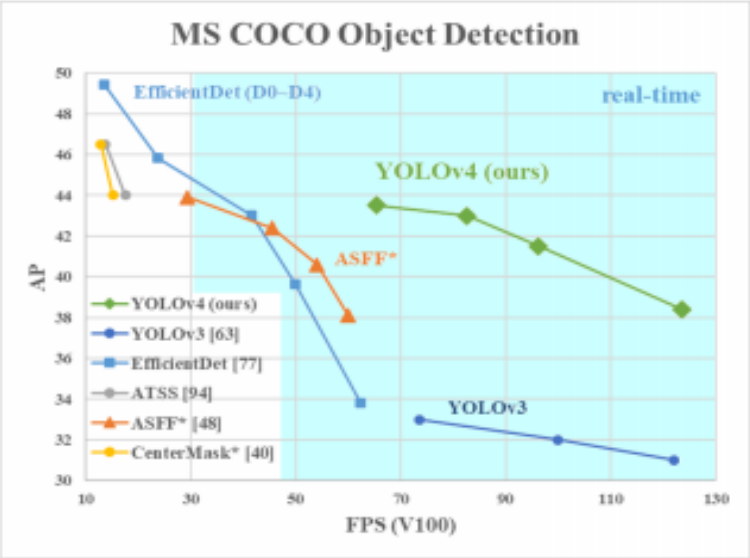

Figura 4.1: Comparación de YOLOv4 frente a otros algoritmos modernos. Obteniendo muchos mejores resultados en cuanto a detecciones en tiempo real con una precisión similar a sus competidores. Figura extraída de [1]

más ligera de YOLOv4.

El proceso para el despliegue en nuestra ZCU104 consta de 4 etapas, la primera conseguir un modelo en Tensorflow en el formato adecuado. A continuación cuantizamos el modelo usando una parte del set de imágenes del entrenamiento. Esto resultará en dos nuevos modelos, uno para su evaluación (comparativas con modelo sin cuantizar) y un segundo modelo el cual compilaremos para desplegarlo en la FPGA usando la API de la DPU.

4.1.1. De DarkNet a Tensorflow

Como mencionábamos brevemente en la introducción anterior, el workflow de Vitis AI esta diseñado para trabajar con modelos desarrollados en TensorFlow (TF) o Caffe. El primer problema es entonces conseguir transformar nuestro modelo en DarkNet a uno de estos dos frameworks. Elegimos TF ya que es el más común y los modelos de ejemplo están desplegados en este.

El modelo en DarkNet consta de dos partes, un fichero *model_name.cfg* que contiene los detalles de la arquitectura de la red y un fichero *model_name.weights* que contiene los pesos y va asociado a un *model_name.cfg* en concreto. En TF cada uno de estos ficheros tienen su equivalente, para el*model_name.cfg* existe un *model_name.pb* y en el caso de *model_name.weights* en TF existe un *model_name.chkpt* pensado para retomar el entrenamiento. Por ultimo debemos fusionar el .pb y el .chkpt en un único *model_name.pb* llamado modelo congelado (frozen model) el cual contiene la información de la arquitectura además de los pesos y se usa en el despliegue de aplicaciones en TF. Nos valdrá para evaluar el modelo en TF además de ser el formato requerido para la cuantización.

Existen varias formas de conseguir el frozen model, pero la recomendada por Xilinx consiste en transformar nuestro modelo a Keras, una API de alto nivel basada en TF y luego transformar el modelo en un frozen model usando el propio Keras y TF. Siguiendo estos pasos preservamos al máximo el modelo en DarkNet de YOLOv4 y no requiere ningún tipo de reentranamiento una vez tenemos el modelo ya en TF. Si quisiéramos hacer ajustes sobre el modelo, como cambiar las clases que podemos detectar o el tamaño del input, esta seria nuestra ultima oportunidad, ya que en el siguiente paso, con el modelo cuantizado el reentrenamiento no es viable.

Antes de seguir verificamos que el resultado sea correcto, usando TF para evaluar el modelo. Para ello será necesario implementar en TF un tipo de capa exclusivo de DarkNet, las capas *yolo*, existen 3 de ellas en el modelo original y se ocupan de transformar el resultado de las ultimas convoluciones en las bounding boxes definitivas. Implica que tras la ejecución del modelo tendremos que hacer un post-procesamiento (Ver 3.1.1), también en el despliegue en la FPGA siempre que queramos evaluar la inferencia. El post-procesamiento es un paso importante y que consume bastantes recursos, además se suele dejar fuera de los cálculos de rendimiento por lo que puede suponer un coste inesperado si no lo hemos tenido en cuenta (ver post-procesado de YOLO en la sección 3.1.1).

4.1.2. Cuantización y compilación

Es en estos dos es pasos donde vamos a usar las herramientas de Vitis AI. Podemos realizar ambos pasos desde un contenedor de Docker desarrollado por Xilinx. En dicho contenedor tenemos las herramientas DECENT (Deep Compression Tool) y DNNC (Deep Neural Network Compiler), que nos serviran para cuantizar y compilar el modelo respectivamente.

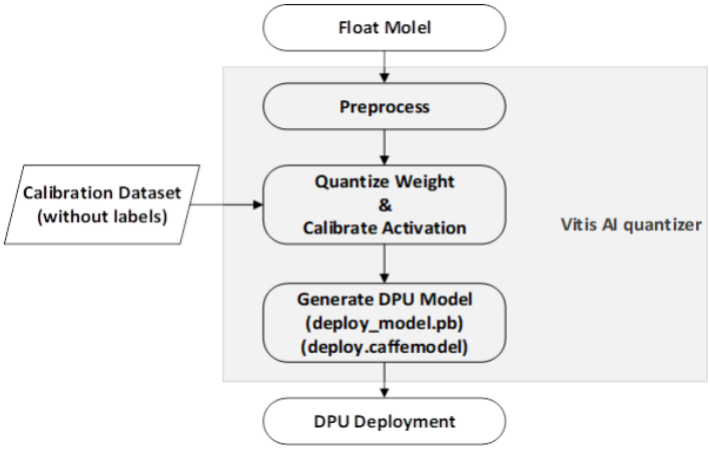

Figura 4.2: Vitis AI Quantizer workflow. Partiendo de un modelo en coma flotante, un set de imagen para la calibración y un algoritmo de pre-procesado se hace uso de esta herramienta la cual genera el *deploy_model.pb* para su posterior compilación y despliegue. Figura extraída de [45]

Después de haber conseguido el frozen model en el paso anterior, DECENT requiere que le pasemos un set de entre 100 y 1000 imágenes, preferiblemente que se hayan usado en el entrenamiento.

Por ultimo necesitaremos una función con la que alimentar las fotografías al modelo, en nuestro caso modificamos la función de ejemplo brevemente para que se ajuste a las necesidades de YOLOv4-Leaky-416. Se puede ver una representación de este workflow en la figura 4.2. El resultado son dos modelos de TF, uno para la compilación con DNNC, *deploy_model.pb* y otro fichero que podemos usar para evaluar el resultado de la cuantización.

Aunque Xilinx no da demasiados datos sobre el proceso de cuantización, es seguro asumir que esta basado en magnitud, es decir, elimina pesos pequeños y de poco impacto, como veíamos en el capitulo 3.2.2.

Con el *deploy_model.pb* generado por DECENT se hace uso del compilador de Vitis AI, DNNC. El resultado es un fichero .elf, precedido del nombre que hayamos decido darle al kernel en el momento de la compilación, en el caso de este trabajo, *yolov4_leaky_416.elf*. El contenido de este fichero es una serie de instrucciones para su ejecución en la DPU, la forma de invocarlas está descrita en la siguiente sección.

4.1.3. Programación con DPU y despliegue

Antes de poder ejecutar el kernel_name.elf son necesarias dos cosas, primero, flashear una imagen en la ZCU104 con la misma versión de la libreria de Vitis AI que la versión con la que se ha compilado el fichero elf. En segundo lugar hay una serie de ficheros auxiliares a los cuales accede la API de la DPU, son el *kernel_name.prototxt* y *meta.json*, el primero define una serie de parámetros en relación al post-procesamiento de la imagen, como el numero de clases, o las capas que producen el output final. El fichero meta.json referencia la librería de la API que se vaya a utilizar al .elf y .prototxt, además de especificar el nombre del kernel del modelo. Ninguno de estos dos archivos se generan en la compilación.

Para poder utilizar el modelo el ultimo paso es generar las tareas de la DPU, usando la ya mencionada API. A la familia de MPSoC Zynq le corresponde la DPUv2 API, y su uso es relativamente simple. Primero creamos el proceso que correrá la DPU en la PL, para ello debemos especificar el nombre del kernel que queramos desplegar y ejecutarlo. El post-procesamiento de la imagen se lleva a cabo desde el PS (Processing Subsystem) y corresponde a los pasos necesarios para convertir las predicciones del kernel en imágenes que se puedan mostrar por pantalla o simplemente mostrarlas en un formato legible. Es por esto que se puede programar en C++ haciendo uso de librerías como OpenCV. La figura 4.3 muestra como podríamos crear un programa de ejemplo.

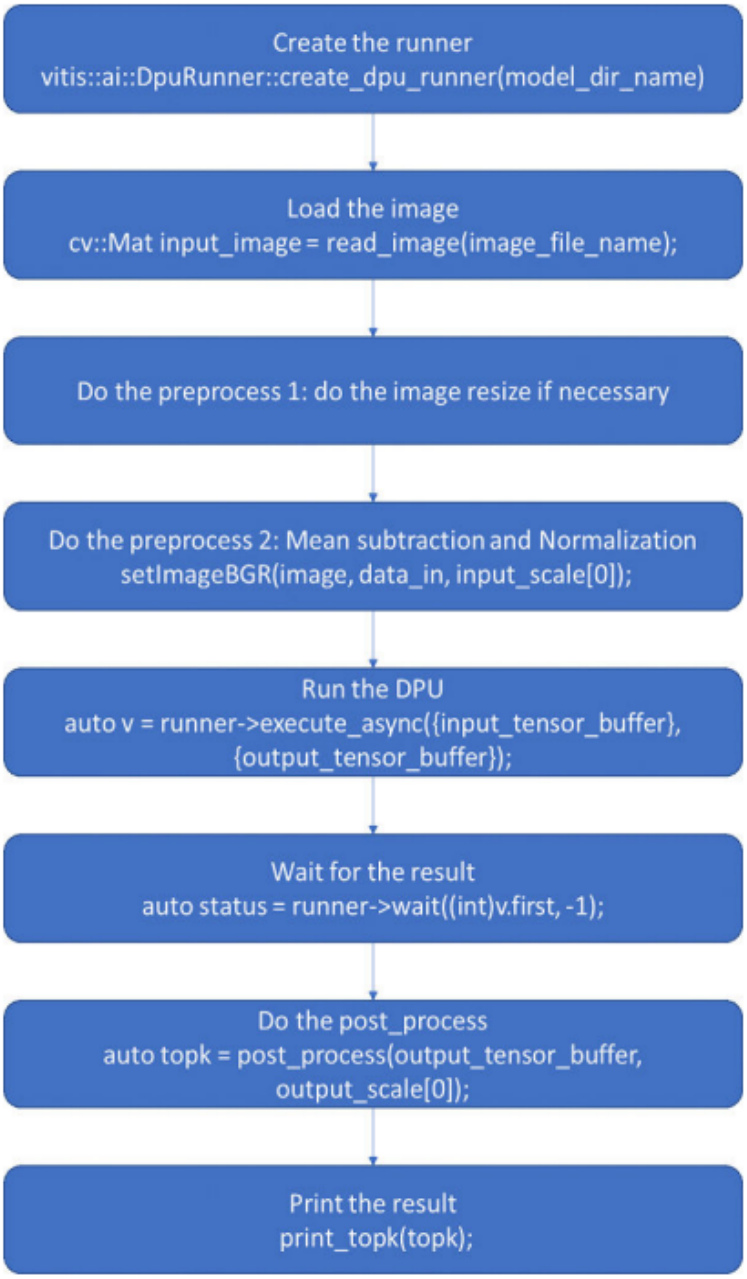

Figura 4.3: Ejemplo de un programa que hace uso de la API DPUv2 para generar predicciones partiendo de un modelo compilado usando el AI Compiler. Figura extraída de [48]

4.2. Entorno de pruebas y métricas

Para poder obtener resultados significativos es importante probar los distintos algoritmos en las mismas condiciones. En este caso probamos la implementación de yolov4-leaky-416 y 608 contra modelos implementados por Xilinx entre los que se encuentra yolov3 (con distintos parámetros) en la versión 1.1 de la librería. Todos en el mismo MPSoc, la ZCU104, usando la versión de la API DPUv2 y usando las mismas primitivas para medir el rendimiento.

A la hora de medir la precisión de un modelo la métrica mas extendida es AP o mAP (Average Precision, mean Average Precision), aunque mAP hace referencia a la media de AP entre clases, se usa AP y mAP indistintamente. Existen varias formas de calcularla, dependiendo del dataset con el que hagamos las mediciones. Esto supone un gran problema a la hora de comparar distintos acercamientos, y es por eso que han surgido intentos de unificar métricas como MLPerf Inference Benchmark [49] o Object Detection Metrics [33]. Este ultimo es el usado en este trabajo para medir el AP de los modelos implementados en el MPSoC, con el fin de tomar las mediciones de la forma mas transparente posible.

En concreto se mide el AP_{50}, métrica usada para evaluar sobre el PASCAL VOC dataset. Donde 50 corresponde con el porcentaje de IoU (entre ground truth y predicción) a partir del cual aceptamos una predicción como correcta (verdadero positivo) o no (falso positivo). Antes de definir AP, es necesario definir especificidad y sensibilidad.

$$Especificidad = \frac{VP}{VP + FP}$$

$$Sensibilidad = \frac{VP}{VP + FN}$$

Ya que comparar detectores usando directamente las curvas de AP es impractico se interpolan todos los puntos de la funcion y se aproxima el area debajo de esta de forma que el AP es la suma de las areas creadas entre cada punto como vemos en la figura 4.4.

Medir el rendimiento es relativamente más fácil. Buscamos métricas para el throughput y la latencia, las mas extendidas son FPS (Fotogramas por segundo, equivalente a inferencias por segundo) y milisegundos en realizar una inferencia respectivamente. Para medir ambas se han usado las primitivas de la API DPUv2. A la hora de calcular el tiempo de cada inferencia se deja fuera el post-procesado. Todas las medidas son utilizando únicamente 1 DPU core. Usar más cores mejoraría el throughput ya que nos permitiría aumentar el tamaño de lote hasta 8 (máximo disponible en la ZCU104) pero no tendría ningún efecto en la latencia.

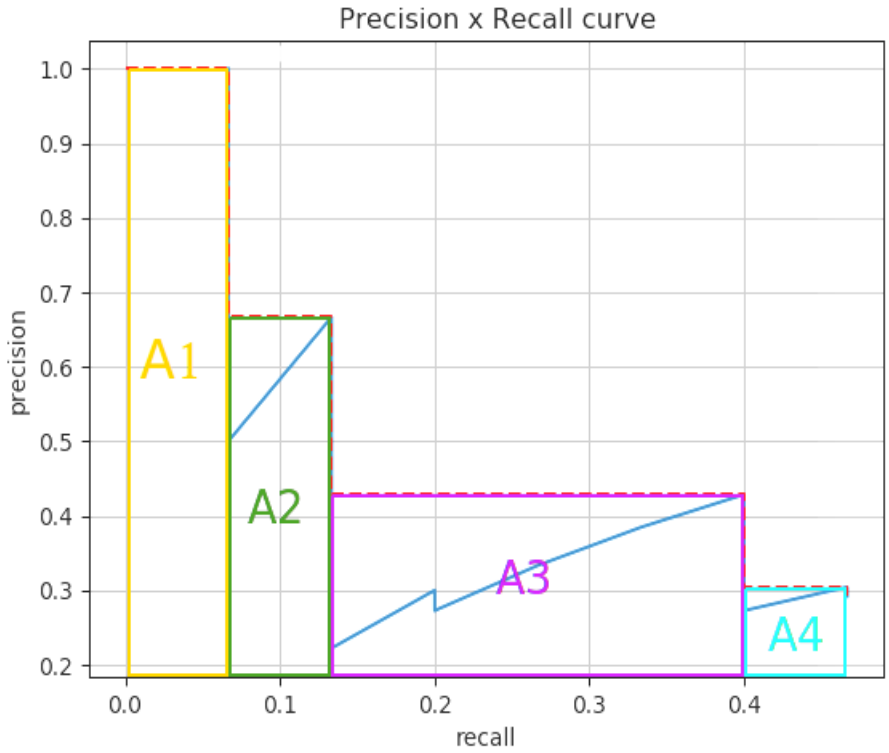

Figura 4.4: En el eje de las *x* tenemos la sensibilidad, en el de las *y* la especificidad. $AP = A1 + A2 + A3 + A4$. Figura extraída de [33].

4.2.1. Arreglo experimental

La figura 4.5 muestra el entorno donde se han llevado a cabo los experimentos ya descritos. Ambos la ZCU104 (4) y el ordenador (3) están conectados a la misma subred. La conexión se realiza mediante *ssh* desde (3). Una vez cargados los ejecutables y los modelos en la ZCU104 los lanzamos desde la consola (2), en el caso de estar probando un vídeo de la cámara es posible reproducir el vídeo, y desde el post-procesado se pintan las bounding boxes, confianza y clase de cada una de las detecciones sobre los fotogramas del vídeo (1) además de imprimirse por terminal (2).

La figura 4.6 nos permite ver de cerca algunos de los componentes principales de la ZCU104 [50]. (1) Corresponde al MPSoC, un chip que combina proccessing system (PS) y PL. El PS esta formado por un Arm Cortex-A53 de 64 bits, 4 núcleos, un Cortex-R5 de dos núcleos encargado del procesamiento de datos en tiempo real y un procesador Mali-400 que maneja el procesamiento de píxeles. La memoria por parte del PS la encontramos en (6), dispone de 4 GB de DDR4-2400. (2), (3) y (4) corresponden a la interfaz ethernet, puerto USB-3.0 y alimentación respectivamente. En (5) encontramos una interfaz micro-SD, en este caso booteamos la ZCU104 desde esta micro-SD, que contiene un sistema operativo basado en Linux. Este sistema está diseñado para el procesamiento de vídeo en tiempo real, dispone de un codec de vídeo integrado, además de soporte para múltiples periféricos e interfaces con un gran ancho de banda.

Figura 4.5: Arreglo experimental. Desde un ordenador (3) con un cliente ssh nos conectamos a la ZCU104 (4), que se encuentra en la misma subred. (1) Muestra el post-procesado de cada uno de los fotogramas que captura la camará conectada a (4). En (2) se van imprimiendo cada una de las detecciones, con su clase, confianza y bounding box coordinates. Las imágenes que se muestran en (1) las toma una cámara conectada mediante USB-3.0 ubicada en el mismo punto desde donde se ha tomado esta fotografía.

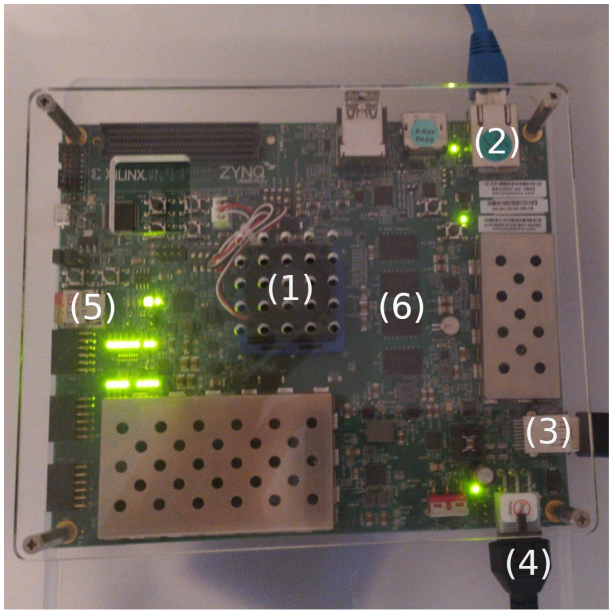

Figura 4.6: Primer plano de la ZCU104. El MPSoC (1) contiene el PS y PL. (2) Es una interfaz ethernet de hasta 1000 Mb/s. (3) Interfaz USB-3.0 soportada por el MPSoC. (4) Corresponde a la toma de corriente. (5) Interfaz micro-SD ademas de una micro-SD de 128 Gb. (6) Memoria de parte del PS, 256 Mb x 16 SDRAMs de DDR4-2400.

4.3. Discusión de resultados

En este apartado se muestran los resultados obtenidos tras medir la precisión y el rendimiento de los modelos desarrollados en este trabajo. La tabla 4.1 compara el despliegue en distintas plataformas. La tabla 4.2 nos permite analizar el rendimiento de cada algoritmo al ser ejecutado bajo las mismas condiciones.

Plataforma	Tamaño	AP_{50}	FPS	Consumo (W)
DPU - ZCU104	416	49.91 %	13.99	27
DPU - ZCU104	608	52.24 %	7.06	28
Titan X Pascal	416	62.8 %	54	260
Titan X Pascal	608	65.7 %	33	260
Tesla V100	416	62.8 %	96	250
Tesla V100	608	65.7 %	62	250
CPU Core i7 7700HQ	Tiny 416	40.2 %	3.4	45
Jetson AGX Xavier	416	62.8 %	9	30

Tabla 4.1: Comparación del rendimiento y precisión de YOLOv4 en distintas plataformas, siendo DPU la plataforma objetivo de este trabajo. Tamaño corresponde con el tamaño del input. Consumo en la DPU corresponde con el consumo máximo alcanzado al ejecutar el algoritmo. Datos extraídos de [1] y el repositorio de Darknet [47].

De la tabla 4.1 podemos sacar las siguientes conclusiones. Las GPUs, en concreto la Tesla V100 se muestra como clara ganadora en cuanto rendimiento, aunque hay que tener en cuenta que el consumo y el precio es un orden de magnitud mayor que el de la ZCU104. Los sistemas empotrados ZCU104 y Jetson AGX Xavier son altamente eficientes en cuanto a consumo y FPS/Vatio como se ve en la figura 4.7. Ambos sistemas trabajan en frecuencias de reloj relativamente bajas y disponen de una memoria que en el caso de la ZCU104 es de 38Mb on-chip. Es entonces donde entran en juego las técnicas de optimización de la sección 3.2 como la cuantización.

Con respecto a sus predecesores YOLOv4 supone una mejora en precisión y eficiencia. Para corroborar esta hipótesis se compara el modelo implementado en este trabajo con distintas versiones de YOLOv3 además de dos variantes de Resnet-50 [2] y MobileNet v2 [44], ver tabla 4.2. Su rendimiento esta a la par de YOLOv3_bdd y YOLOv3_voc_tf, teniendo en cuenta que puede detectar 8 y 4 veces más clases de objetos respectivamente.

Para poner en perspectiva YOLOv4_leaky con otros modelos entrenados usando el mismo dataset se ha probado el rendimiento de ssd_resnet_50_fpn_coco_tf y ssd_mobilenet_v2_coco_tf. La implementación de YOLOv4_leaky_608 con un tamaño de input muy similar a ssd_resnet_50_fpn_coco_tf tiene un rendimiento casi 6 veces mayor, en parte gracias al rápido post-procesado de YOLO. Se aprecia la importancia de elegir un determinado tamaño de input, dada su influencia en el rendimiento. Y

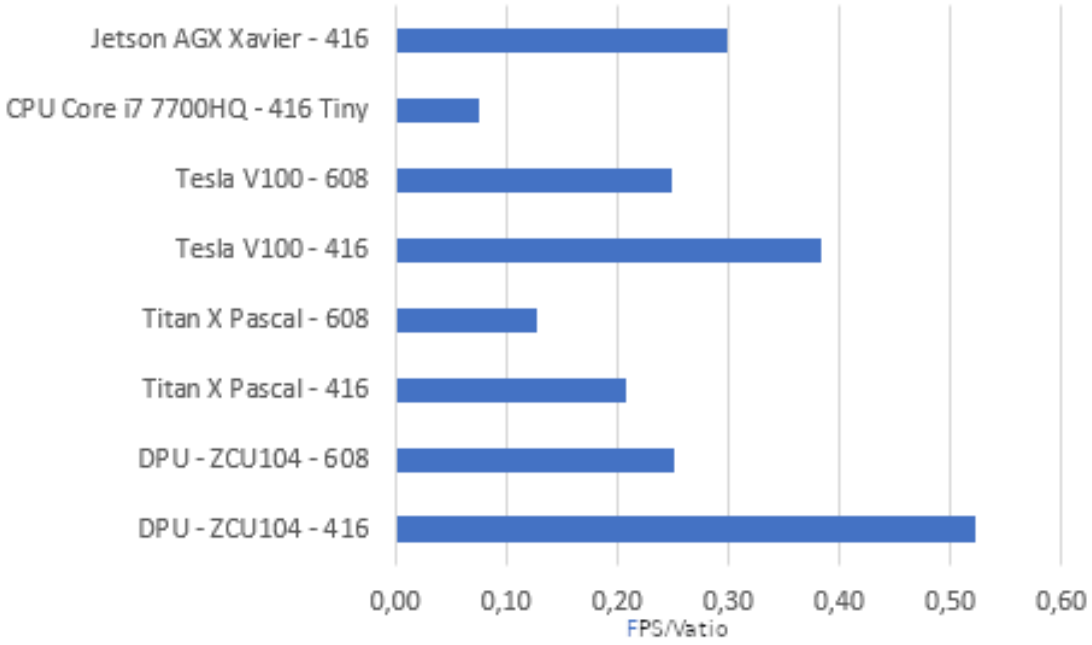

Figura 4.7: Comparativa de las distintas plataformas y modelos, eje de las *x*, con respecto a FPS/-Vatio en el eje de las *y*. El modelo más eficiente, DPU-ZCU104 con un input size de 416, hace uso de las distintas técnicas de optimización descritas en este trabajo.

Modelo	FPS	Inf. DPU (ms)	Inf. E2E (ms)	Tamaño Input	Clases
YOLOv4_leaky_416	13.9	67.7	71.3	416x416	80
YOLOv4_leaky_608	7	135.3	141.4	608x608	80
YOLOv3_bdd	13.5	70.6	73.4	512x288	10
YOLOv3_voc_tf	14.1	68.5	70.6	416x416	20
YOLOv3_adas_prunned	85.4	9.2	11.7	512x256	3
ssd_resnet_50_fpn_coco_tf	1.3	171.6	749.9	640x640	80
ssd_mobilenet_v2_coco_tf	38.5	9.2	25.9	300x300	80

Tabla 4.2: Rendimiento de YOLOv4 comparado con otros algoritmos ejecutados en la misma ZCU104 usando la API DPUv2 de Vitis. Las medias se han tomado sobre el mismo set de imágenes perteneciente al set de evaluación de COCO 2014. E2E (End to End) corresponde con una inferencia teniendo en cuenta post/pre-procesado, y es la que se tiene en cuenta a la hora de calcular los FPS. Para entender las diferencias entre rendimientos es importante tener en cuenta el impacto del tamaño del input y las clases detectadas.

como se mostraba en la tabla 4.1 la ganancia en precisión no parece ser tan notable si el objetivo es un detector en tiempo real. Es por esto que modelos como YOLOv3_adas_prunned, diseñado para asistencia a la conducción, elige un tamaño de input 300x300, sacrificando en gran parte precisión por rendimiento.

5

Conclusiones y trabajo futuro

Por ultimo en este capitulo se exponen las conclusiones fruto de haber desarrollado este trabajo. Ademas se discuten diferentes formas de las que se podría continuar y expandir sobre el sistema implementado.

5.1. Conclusiones

Comenzábamos este trabajo describiendo una serie de objetivos, entre ellos medir rendimiento y precisión de una versión nueva de un algoritmo ampliamente conocido como es YOLO. Al comenzar el desarrollo de este trabajo YOLOv4 aún no había sido implementada de forma publica en una plataforma de Xilinx como es la ZCU104. No estaba claro ni siquiera si llegaría a funcionar. No solo se ha comprobado su viabilidad, sino que se ha podido corroborar que YOLOv4 supone una mejora en cuanto a detección de objetos en tiempo real comparado con su predecesor YOLOv3 y algoritmos similares. Además de probar la eficiencia energética de la plataforma sobre la que se ha desplegado frente a GPUs y CPUs. Los objetivos teóricos también los consideramos cumplidos. Usando un acercamiento de arriba abajo ha quedado retratado el estado en el que se encuentra la visión por computador y como conocer en detalle el funcionamiento de los algoritmos nos puede ayudar a mejorar su rendimiento a la hora de ejecutarlos.

De forma indirecta también se ha probado la eficacia de las herramientas de Vitis AI. Esta tarea, en la que se ha implementado una red neuronal en un sistema empotrado de forma eficiente, sería impensable usando únicamente herramientas de desarrollo de hardware más clásicas, y menos por un estudiante con tan poca experiencia en el diseño de hardware.

Para concluir, cabe destacar el esfuerzo y la cantidad de prueba y error que ha requerido cada uno de los pasos de la implementación. Ha sido necesario aprender sobre muchas y muy diversas tecnologías, desde frameworks de aprendizaje automático a diseño de hardware. Y aunque en el grado no se enseñen directamente muchas de estas herramientas, la metodología y los conocimientos adquiridos estos últimos años han sido claves para poder realizar este trabajo.

5.2. Trabajo futuro

Durante el desarrollo se han necesitado numerosos recursos, desde una FPGA a un ordenador suficientemente potente para realizar tareas de cuantización y compilación. Y aunque hemos usado únicamente software libre, existen herramientas como AI Optimizer de Vitis AI el cual solo tiene acceso bajo licencia. Expandir los recursos en cuanto a cómputo y licencias haría posible aumentar el abanico de técnicas de optimización utilizadas (como el pruning), e incluso entrenar nuestro propio modelo con el fin de ajustarlo a una tarea concreta.

Otra extensión lógica consistiría en implementar un sistema empotrado de altas prestaciones en una aplicación real. Como podría ser un coche autónomo o un robot asistencial y medir su viabilidad frente a otros acercamientos.

En cuanto a las métricas, sería interesante estudiar el impacto individual de cada una de las técnicas aplicadas. Esto prueba ser una tarea bastante costosa ya que requiere medirlas por separado una a una y dadas limitaciones de tiempo no ha sido posible hacerlo en este trabajo. También se contempló hacer un estudio más exhaustivo de las métricas de precisión. Esto ha probado ser muy complicado causa de la falta de consenso en cuanto a métricas y la falta de transparencia que existe sobre las condiciones en las que se toman las medidas (Por parte de fabricantes de hardware y desarrolladores de software).

Bibliografía

[1] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[2] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," *arXiv preprint arXiv:1602.07261*, 2016.

[3] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[4] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[5] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[7] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[8] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[10] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[11] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[12] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 2, pp. 19–227, 2020.

[13] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[15] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*, 2010.

[16] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, p. 3, 2013.

[17] M. Gschwind, V. Salapura, and O. Maischberger, "Space efficient neural net implementation," in *Proceedings of the 2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[18] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[19] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014.

[20] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[21] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD: single shot multibox detector," *CoRR*, vol. abs/1512.02325, 2015.

[22] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.

[23] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," 2017.

[24] L. Najman and M. Schmitt, "Watershed of a continuous function," *Signal Processing*, vol. 38, no. 1, pp. 99–112, 1994.

[25] R. Nock and F. Nielsen, "Statistical region merging," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 26, no. 11, pp. 1452–1458, 2004.

[26] B. Settles, "Active learning literature survey," tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 2009.

[27] Y. Siddiqui, J. Valentin, and M. Nießner, "Viewal: Active learning with viewpoint entropy for semantic segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9433–9443, 2020.

[28] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3213–3223, 2016.

[29] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *arXiv preprint arXiv:2001.05566*, 2020.

[30] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440, 2015.

[31] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[32] Itseez, "Open source computer vision library." https://github.com/itseez/opencv, 2015.

[33] R. Padilla, S. L. Netto, and E. A. B. da Silva, "A survey on performance metrics for object-detection algorithms," in *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*, pp. 237–242, 2020.

[34] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[35] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, and Y. Xu, "Throughput-optimized fpga accelerator for deep convolutional neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 3, pp. 1–23, 2017.

[36] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[37] M. Denil, B. Shakibi, L. Dinh, M. A. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems 26* (C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, eds.), pp. 2148–2156, Curran Associates, Inc., 2013.

[38] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, 2017.

[39] S. R. Jain, A. Gural, M. Wu, and C. H. Dick, "Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks," *arXiv preprint arXiv:1903.08066*, 2019.

[40] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, pp. 1135–1143, 2015.

[41] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5687–5695, 2017.

[42] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[43] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 285–300, 2018.

[44] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[45] Xilinx, "Vitis ai user guide." https://www.xilinx.com/support/documentation/sw_manuals/vitis_ai/1_0/ug1414-vitis-ai.pdf, 2020.

[46] Xilinx, "Dpu for convolutional neural network v2.0." https://www.xilinx.com/support/

documentation/ip_documentation/dpu/v2_0/pg338-dpu.pdf, 2020.

[47] AlexeyAB, "Darknet." https://github.com/AlexeyAB/darknet, 2020.

[48] Xilinx, "Vitis ai library user guide." https://www.xilinx.com/support/documentation/sw_manuals/vitis_ai/1_1/ug1354-xilinx-ai-sdk.pdf, 2020.

[49] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," 2020.

[50] Xilinx, "Zcu104 evaluation board user guide." https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf, 2020.